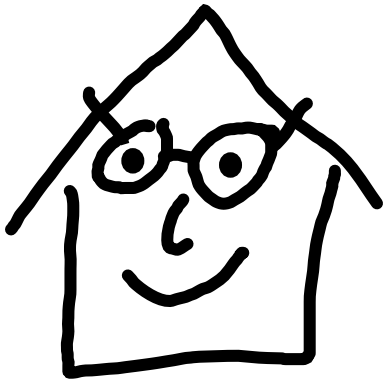


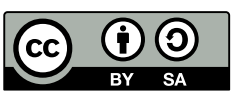
The *Home2L* Book

Clever Tools for a Private Smart Home

Gundolf Kiefer

Version: v1.0-2-ge04a

March 3, 2020



Contents

1. Introduction

1.1. What Are the *Home2Ls*?

The *Home2L* [houmtu:l] suite is a framework, library and set of tools for automation in private smart homes. Its main features and design goals are:

Efficient and Lightweight Design

All core components are written in C/C++, with a very minimum set of external dependencies beyond *libc* - ideally suited for small embedded devices and microcontrollers. There is no need for a Java runtime environment or a heavy web framework. Starting up a server and a command shell and shutting both down again takes less than a second altogether - on an ARM-based minicomputer running at 144 MHz!

Ambient Intelligence, No Need for a Central Server

Central servers are single points of failure. *Home2L* follows a completely distributed concept. Any (mini-)computer can act as part of the network. If resources, such as sensors or actors, are connected to them, they can be exported to any other host in the *Home2L* network. A failure of a host only causes its own resources to be unavailable – everything else keeps on working.

Automation Rules Written in Python – But Not Limited to That

There is no new language or tool to learn to formulate automation rules. *Home2L* rules are typically formulated in Python, they profit from the simplicity and power of the Python language. There can be multiple rules scripts, they may run on any machine, and they may be combined with other software routines or be part of a larger application.

Other ways to interact with *Home2L* resources is via the C/C++ API from any application or by shell scripts using the *Home2L Shell* in non-interactive mode.

Easy Driver Development in C/C++, Python or Any Other Language

An API for *resource* drivers allows to easily add support for new hardware. A driver can be implemented in C/C++, in Python, or as a shell script. For all these cases, documented examples can be found in the source tree (see Section 5.8).

Easy Integration of "Do-it-Yourself" Hardware

Home2L Brownies are programmed low-cost microcontrollers (*AVR ATtiny 84/85/861*) connected to a Linux host over 4-wire cables (e.g. KNX/EIB cables). The bus protocol is based on *i2c*, robust and allows to build very simple, self-made hardware nodes. Just an *ATtiny* device and two resistors are enough to build a sensor node!

Privacy

The *Home2Ls* do not need an Internet connection and do not try to communicate with hosts other than they are configured to. By design, the *Home2Ls* communicate with each other over a (trusted) LAN, which can easily be set up and secured using standard Linux/UNIX techniques. The open source licensing ensures transparency for what the software does inside the user's private home.

Modularity

The core part, the *Resources* library, is kept small and portable with APIs for application programs and drivers in C/C++ and Python. All other components are optional and can be used or replaced by alternatives as desired by the user.

Figure 1.1 shows an overview on the *Home2L* suite. It contains a collection of mutually independent applications (blue). The central component is the *Home2L Resources* library, which is used by all applications. Resource drivers (brown) are also optional and provide interfaces to sensors, actors or services like *MQTT* brokers.

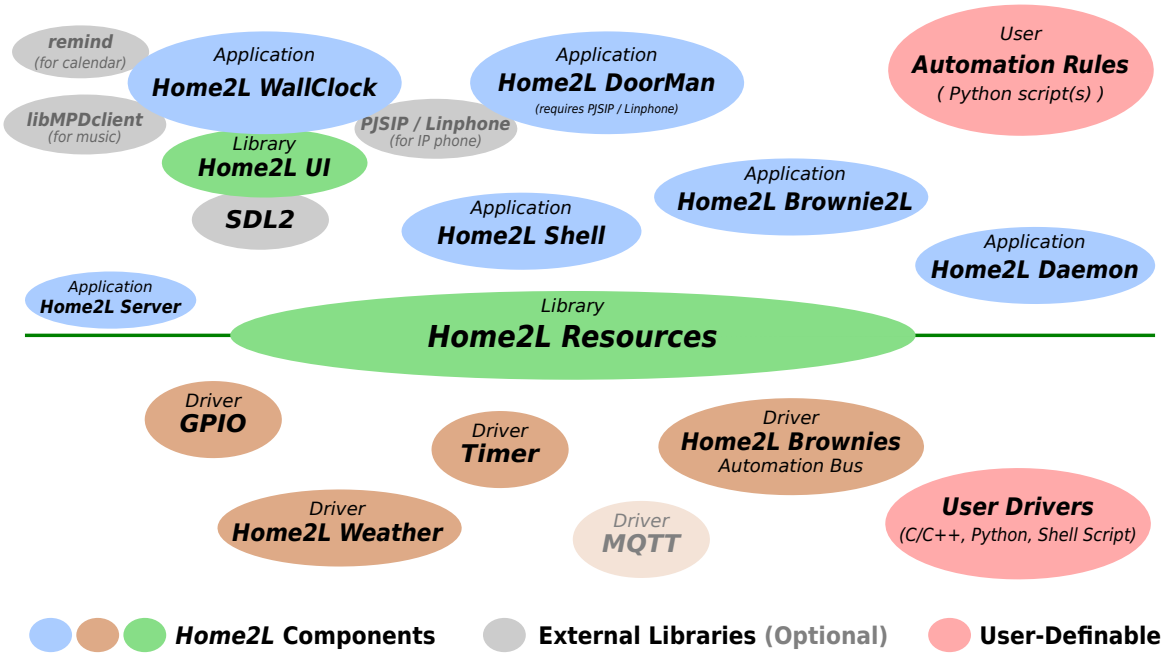


Figure 1.1.: *Home2L* Components: Tools, Drivers, Libraries

1.2. About This Book

This book serves as the main user and reference manual for the *Home2L* suite.

Further information can be found in the C/C++ API and the Python API documentation. Consulting the API documentation is particularly helpful for writing own resource drivers, sophisticated automation rules or applications using some *Home2L* library.

Presently, the *Home2L Book* is still under construction and primarily targets readers with computer skills. Not all existing features are already documented in this book. Extending the book to provide good end-user documentation is subject to future work (see Section 1.3).

The *Home2L* suite comes with a Docker Demo Image. Chapter 2 contains a step-by-step tutorial based on this image demonstrating the core concepts and features of the *Home2L* suite. This should be the starting point for any new user.

Chapters 3 and 4 cover the installation and administration of a *Home2L* cluster.

Chapter 5 explains the concepts and usage of the *Resources* library, the core component of the *Home2L* suite. This is followed by explanations on how to write automation rules in Chapter 6.

Chapter 7 introduces the *Brownies* subproject. This includes a description of the physical properties and the protocol of the *Brownie* automation bus, an introduction to the software stack on the microcontroller and on the Linux host side, as well as example circuits.

The following chapters cover the main user applications, presently the *Home2L WallClock* (Chapter 8) and the *Home2L DoorMan* (Chapter 9).

Chapter 10 documents the library of drivers supplied with the *Home2L* base distribution.

Links in this document

This document makes extensive use of links pointing into the code documentation or the source tree to help finding relevant information quickly. Some hints on using these links:

- Some links name a code object in the text (e.g. a function, class or macro), and the links refer to the root of the C/C++ API or the Python API documentation. In such a case, you can usually copy the name of the code object and paste it into the "search" field of the code documentation web page to quickly navigate to the respective code.
- Some links point to some file or directory in the source tree. Due to a bug in the LaTeX "hyperref" package, it appears to be impossible to refer to a file or directory without a period in its name. As a workaround, some link targets end with an extra period (".."). Please remove an eventual trailing period if it occurs in a link target.

1.3. How Can I Help?

Until now, the *Home2L* project has been developed by a single private person in his spare time. The code has been published with the hope that is useful to the community.

To let the project grow further and make available to a wider audience, volunteers and partners are needed and welcome.

Great contributions would be:

1. **Report and help fixing bugs.**
2. Make **sample installations**, document them and share your experiences.
3. **Packaging**: Create packages for major Linux distributions.
4. **Bridges to other home automation frameworks:** Implement drivers to interface with other open home automation frameworks.
5. **More drivers:** Implement drivers for any hardware you have or like. Help with an *MQTT* bridge would be appreciated, too.
6. **Documentation**: Write good documentation, particularly for end users.

For any questions on how to participate, do not hesitate to contact the author via the project page.

2. Tutorial: Have a Test Drive!

2.1. Introduction

2.1.1. Overview

This step-by-step tutorial aims to give an introduction to the *Home2L* suite covering all core tools and concepts of the suite. It should be the starting point for anybody new to the software.

This tutorial will introduce you to:

- the main *Home2L* concepts and capabilities.
- *Home2L Resources*: values and states, subscriptions, requests, the directory.
- the *Home2L Shell* for inspection, maintenance, and logging.
- *Home2L Brownies* and the integration of do-it-yourself hardware.
- the *WallClock* UI, including the music player and family calendar.
- writing automation rules using the Python API.
- several command line tools: `home2l-shell`, `home2l-wallclock`, `home2l-daemon`, `home2l-server`, `home2l-brownie2l`.
- many code examples for automation rules and drivers.

The tutorial can be performed using a prepared Docker container image with a complete showcase demo setup. No manual installation is necessary.

Sections 2.2 – 2.4 introduce the showcase system and basic concepts of the *Home2L* suite. The remaining subsections cover dedicated topics and can be exercised independently from each other, depending on your interests and preferences.

2.1.2. Prerequisites

To run the tutorial you need

- Docker (Version 18.09.01 or above),
- a host operating system with a running X11 server (e.g. Linux).

Alternative ways of running the tutorial without the supplied Docker image are described in Section 2.10.

2.1.3. Notational Conventions

The tutorial uses the following conventions in notation:

a) A black triangle (▶) marks an instruction to follow.

b) `Typewriter text with a grey background` marks an interaction with your computer. Lines starting with a prompt indicate commands to be entered into the respective interpreter:

`$` – the Linux shell (*bash*),

`home2l>` – the *Home2L Shell*,

`>>>` – the Python command interpreter.

Example

▶ Enter the following commands to verify your computer's calculation skills:

```
$ python3
>>> 2+3
5
```

Then push *Ctrl-D* to quit the Python shell again.

Info and Warning Boxes

i | Info boxes contain some additional information, not essential to just run the tutorial successfully.

⚠ | Warning boxes contain important information. You should read it, otherwise the tutorial may not work as expected, or some other kind of problem may occur.

2.2. Starting the Showcase Environment

▶ On your host PC, open a terminal window and enter:

```
$ xhost +local:        # allow X11 applications in the container to open windows
$ docker run -ti --rm --name home2l-showcase --hostname home2l-showcase \
    -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix --device /dev/snd \
    gkiefer/home2l
```

That's it!

You should now have two windows open as shown in Figure 2.1. The left window is your original terminal window, in which a console application, `home2l-showhouse`, is running. It simulates the house and the physical world. The right window shows a `home2l-wallclock` instance, which simulates a virtual tablet computer mounted at the living room wall of your virtual home.

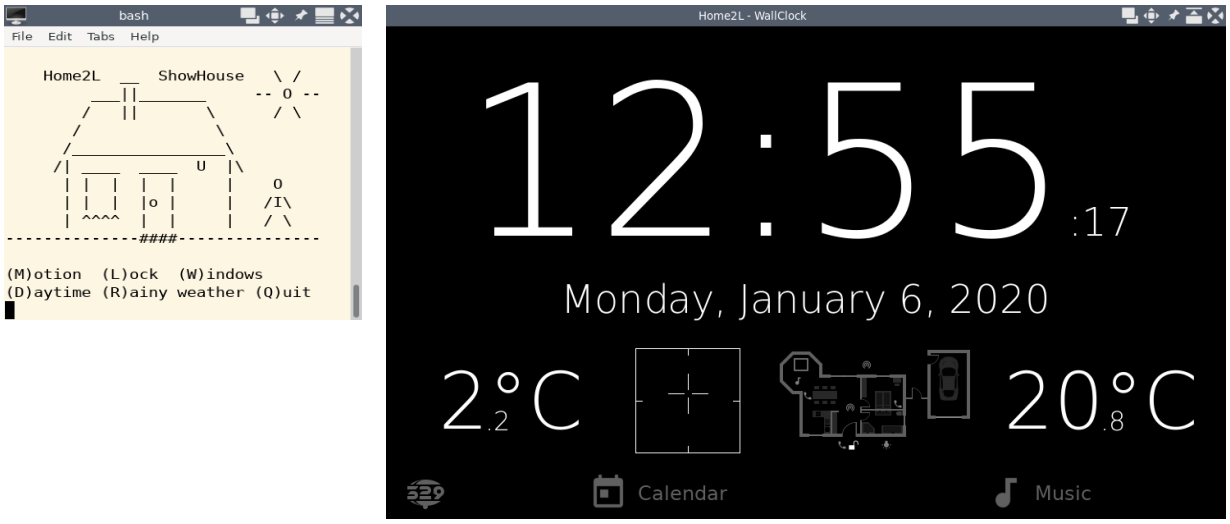


Figure 2.1.: Open windows after starting the showcase environment

i

The Docker container is running several foreground and background processes (to be explored later). In future steps, you may want or need to start additional programs inside the running container. This can be done by running (in some other terminal window):

```
$ docker exec -ti home2l-showcase <command>
```

It is recommended to define aliases for running commands normally or with *root* privileges, respectively:

```
$ alias DOCKER="docker exec -ti home2l-showcase"
$ alias DOCKER_ROOT="docker exec -ti --user=root home2l-showcase"
```

Examples:

```
$ DOCKER bash           # Run a (Unix) shell
$ DOCKER home2l shell   # Run home2l-shell
$ DOCKER_ROOT bash      # Run a root shell
```

2.3. Exploring the *ShowHouse*

Now that the *Home2L* showcase environment is up and running, it is time to enter the virtual *ShowHouse* and look around a little bit.

Purpose of this section is to

- give an overview on the showcase system,
- introduce basic *Home2L* features from a user perspective,
- demonstrate the distributed design paradigm and failure tolerance of the *Home2Ls*,
- give a first impression of the power and capabilities of the *Resources* library.

2.3.1. Overview

The *Home2L* suite follows a highly distributed design paradigm. At this point, there are 4 tools (*Home2L instances*) running on your machine – two are visible, two others running in the background:

1. The *ShowHouse* (`home2l-showhouse`, left in Figure 2.1) is a simulator for the physical world of the demo system. This little console application contains a *Resource* driver providing a simulated motion sensor, a door lock sensor, window shades, a door light and some other gadgets. The user (you) can simulate physical actions by key presses, and some gadgets are visualized in the ASCII art picture.
2. The *WallClock* (`home2l-wallclock`, right in Figure 2.1) is a universal information display to be mounted on room walls or installed on mobile devices. This instance is (virtually) mounted in the living room of the (virtual) house.
3. A `home2l-server` instance is running in the background, which in this tutorial hosts the weather driver (`home2l-drv-weather`).
4. A background script (`rules-showhouse`) executes various automation rules, for example to close the shades at night or to automate the outdoor light. This script will be explored later in Section 2.5.

The following subsections explore this setup step by step, beginning with the most visible instances.

> ***i*** **Do the *Home2Ls* need a central server?**
>
> No. Even if there is an instance `home2l-server` whose name suggests so, a *Home2L* installation does not need a central server. In this example, the weather driver could as well be loaded and executed by *any* of the other three *Home2L* instances.
>
> On the other hand, in a larger installation, it may be useful to run multiple *Home2L* server instances in order to improve failure resilience, to attach sensors/actors to different machines, or for maintenance reasons.
>
> The assignment of the *weather* driver to the *server* instance is configured in the `home2l.conf` configuration file and may be changed there.

2.3.2. The *WallClock* Home Screen

The *WallClock* display shows the current time and date in its upper area. The bottom row contains launcher buttons to access the integrated calendar and music applets (to be explored later).

> **Resizing the *WallClock* window**
>
> The *WallClock* window can be resized arbitrarily, and the UI is scaled automatically to fit into the window. Alternatively, the window can quickly be resized to reasonable sizes by pushing *F9* (half size), *F10* (normal) and *F11* (fullscreen), respectively. By pushing *F12*, the size can be fixed (may be useful when using a tiling window manager like *awesome*).
>
> For the tutorial, it is best to keep it in its normal size (*F10*).

In the lower part of the display between the date and the launcher buttons, the values of several sensors are visualized. These are, from left to right:

1. the local outside temperature,
2. the *radar eye* – a visualization of weather radar around the building (the location is pre-configured for Augsburg, Germany),
3. the mini floor plan,
4. the room temperature.

Each of these fields visualizes some or multiple *Home2L Resources* provided by different sources.

> **If the outside temperature or radar eye are missing ...**
>
> ... don't worry!
>
> The weather driver depends on an available internet connection and on the *OpenData* server of the German Weather Service (DWD). If the server is not reachable this time, the weather-related items are missing in the *WallClock* display. If so: Do not worry, the other parts of the tutorial are not affected.
>
> You can diagnose the availabilty of weather data by running the weather driver directly in a terminal:
>
> ```
> $ DOCKER bash
> $ . /opt/home2l/env.sh
> $ /opt/home2l/lib/home2l-drv-weather weather.debug=1
> ```

2.3.3. The *WallClock* Floor Plan

By pushing (or clicking on) the mini floor plan, you can open the full floor plan view (see Figure 2.2). The buttons on the bottom of the display are mainly used to set the presence (use) state of the user (from left to right):

1. "Back" button to return to the home screen,

2. "Auto" button to let the presence state be set automatically based on the daytime.
3. User is at home at day time.
4. User is at home and sleeping.
5. User is away temporarily (e.g. for work).
6. User is away for longer (e.g. on vacation).

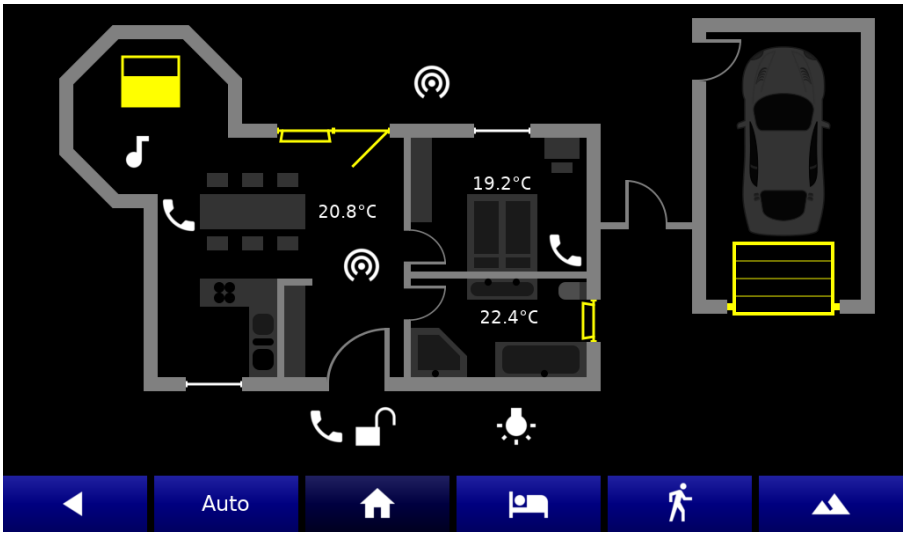


Figure 2.2.: The *WallClock* floor plan

2.3.4. The *ShowHouse*

Now it is time to explore the *ShowHouse* console tool by simulating some physical user actions and weather events.

- In the *WallClock* window, push (or click on) the mini floor plan to open the full floor plan view.

  Then focus the *ShowHouse* window.

- Press the keys listed in the *ShowHouse* window and see how the various gadgets (daylight, door lock status, motion, ...) are visualized in the ASCII art and in the floor plan.

  Did you notice that ...

  - the shades close automatically at night?
  - the door light is activated for a few seconds if motion is detected (press 'M'), but only at night time?
  - at night time, open windows or an unlocked main door are highlighted to draw the users' attention to it?
  - during rainy weather, the *WallClock* excitedly reminds you to close open windows?

The first two points are controlled by the rules script discussed later in Section 2.5.

- In the *WallClock* window, return to the home screen (push the "back" button). Then repeat the previous step to see that most gadgets are also visible in the mini floor plan.

  What happens, if it starts to rain while a window is open?

2.3.5. Interacting with the *WallClock* UI

Now let us explore how we can control various gadgets with the *WallClock* UI.

- In the *WallClock* window (floor plan screen), click on the roof window, close the shades and open them again. You may also close them partially. You can do the same with all the shades of the other windows.
- The shades dialog has an "Auto" button, which lets you decide whether the shades should be controlled manually or not. Verify this with the shades of the bedroom window. Open and close it manually and check if it is still opened/closed automatically at day or night time (press 'D' in the *ShowHouse* window).
- Push the door light icon to switch the light on and off manually. Similar to the shades, the dialog has an "Auto" button and lets you decide whether the light should be forced in an on/off state or whether automatic rule shall control it. By pushing long on the icon, you can toggle between the "on" and the "auto" state.
- Click on one of the WiFi icons to switch on the (virtual) WiFi access points. Then open some windows and unlock the main door.

  Now assume, you have to leave the house for a short (or a longer) period of time. Click on the "temporarily away" (or "away for longer") button in the bottom right of the screen. See how the system reminds you to close the windows and eventually switch off the access points before you leave!

  Finally, click on the "Auto" button to indicate that you are back home.
- Feel free to look around and try other features!

2.3.6. Is All This Stable and Robust? – Failure Resilience

The following steps examine how the system behaves if one or several *Home2L* instances fail. Failure handling and reporting is provided by the *Resources* library to facilitate application development.

- In the *WallClock* window, switch back to the home screen. Then focus the *ShowHouse* window.
- Quit the *ShowHouse* (press "Q") and see the mini floor plan and the room temperature disappearing in the *WallClock* display. These resources were (mainly) provided by the *ShowHouse* process, which is now no longer running.
- Stop the background processes (`home2l-server` and `rules-showhouse`) by entering:

```
$ home2l demo stop
```

The weather information displays disappears, and the automation rules stop working.

- Start the *ShowHouse* again:

```
$ home2l showhouse
```

The floor plan and the inside temperature appear again.

Toggle between day and nighttime (press "D") and simulate motion at night (press "M"). The light is not turned on and the shades do not close at night, because the automation rules script is not working!

- Finally, quit the *ShowHouse* (press "Q") and start everything again:

```
$ home2l demo start
$ home2l showhouse
```

In summary, these examples show that whenever a *Home2L* instance fails, only the resources or functionalities of failing instance stop working. Everything else remains available. There is no single point of failure. By the remaining tools, the absence of a resource is detected reliably, in many cases instantaneously.

2.4. Looking Behind the Scenes – Using the *Home2L Shell*

The *Home2L Shell* (`home2l-shell`) is the command line interface to the *Resources* library and serves as a "swiss army knife" to access and inspect the resources and servers of a *Home2L* cluster. With the *Home2L Shell* you can inspect all servers and list and inspect all resources. Furthermore, it can be used in a batch mode in a very flexible way for tasks like data logging, manipulating resources from shell scripts or for testing new drivers.

This section gives an introduction to the *Home2L Shell*, which we will use to take a look behind the scenes of the *ShowHouse*. In particular, we will use the shell to check the status of all *Home2L* instances, explore the directory tree, manipulate and monitor resources, and use the *Home2L Shell* for data logging.

2.4.1. Getting Started

- To simplify the following steps, it is recommended to define an alias for running commands in the currently running Docker container.

  Open a new terminal window and enter:

  ```
  $ alias DOCKER="docker exec -ti home2l-showcase"
  ```

  Make sure that all three windows – the *WallClock*, the *ShowHouse* window and the new *Shell* window are visible for you. See Figure 2.3 for a recommended screen layout.

- In the *Shell* window, start the *Home2L Shell*:

  ```
  $ DOCKER home2l shell
  ```

  The shell has an integrated help functionality. Type

  ```
  home2l> h
  ```

  to get a list of available commands or

  ```
  home2l> h <command>
  ```

  to get more detailed information about a certain command.

- Check the status of all servers in the *Home2L* network:

  ```
  home2l> n
  server          (    127.0.0.1:4701): OK, standby (since 2020-01-11-135911)
  showhouse       (    127.0.0.1:4700): OK, standby (since 2020-01-11-135911)
  wallclock       (    127.0.0.1:4702): OK, standby (since 2020-01-11-135911)
  ```

  The list indicates that all servers are OK and reachable from the *Home2L Shell*.

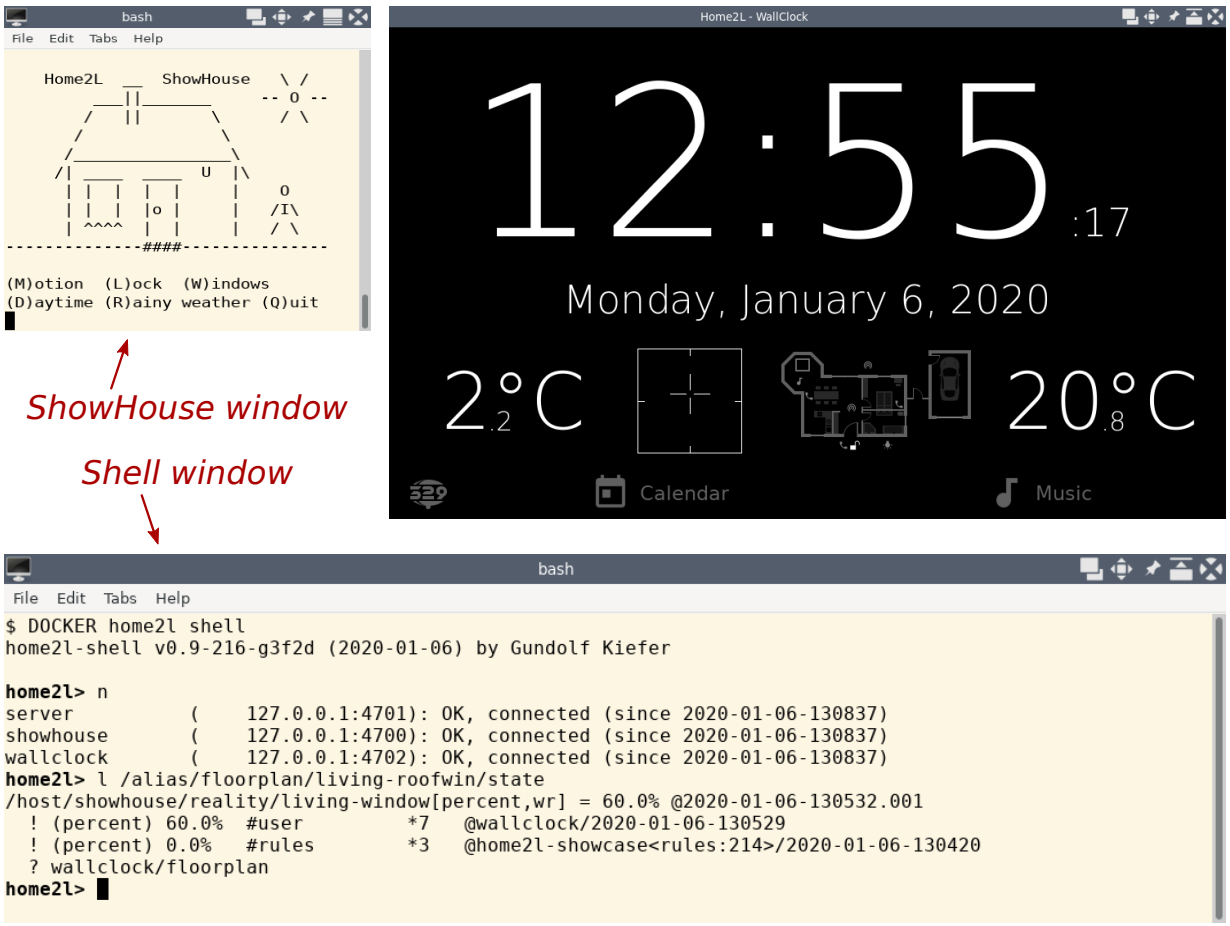


Figure 2.3.: Recommended screen layout

2.4.2. Navigating and Inspecting Resources

- Use the commands 'c' and 'l' to explore the available resources and the namespace. Tab-completion is available. List the root directory of the namespace:

```
home2l> l /
alias
env
host
local
```

The directory */host* contains an entry for each host:

```
home2l> l /host/
home2l-showcase<shell:431>
server
showhouse
wallclock
```

The directory */alias* contains all defined alias names, which are symbolic links to host resources or directories:

```
home2l> l /alias/
daylight -> /host/showhouse/world/simDaylight
floorplan/bath-shades/shades -> /host/showhouse/world/bath-shades
floorplan/bath-temp/data -> /host/showhouse/signal/fp_temp_bath
floorplan/bath-window/state -> /host/showhouse/world/bath-window
...
use -> /host/wallclock/signal/use
weather -> /host/server/weather
weatherWarning -> /host/showhouse/world/simBadWeather
```

The top-level directory */local* is a hard-wired alias to */host/<self>*, where *<self>* is the local *Home2L* host ID (*"home2l-showcase<shell:431>"* in this case).

- With the 'c' command, you can navigate in the tree.

```
home2l> c /alias/frontLight
/alias/frontLight
```

> ***i*** If you are familiar with navigating and exploring files using "cd", "ls" and tab-completion in a Unix shell: Navigating with the *Home2L Shell* is very similar, just easier!
>
> a) The respective commands have one character instead of two ("c"/"l" instead of "cd"/"ls").
>
> b) Tab-completion may auto-complete over multiple directory levels.
>
> c) The "l" command does not only list directories, but displays anything, particularly resources. Similililarly, with "c" you can not only navigate to a directory, but also to a resource itself.

- The "l" command without arguments lists the current directory or object. Running it now displays the status of the outdoor light resource.

```
home2l> l
/host/showhouse/world/light[bool,wr] = 0 @2020-01-11-135724.842
  ! (bool) 1         #motion       *3 -2020-01-11-140606.472   @home2l-showcase<rules↩
    ↪:359>/2020-01-11-140601
  ! (bool) 0         #daytime      *4   @home2l-showcase<rules:359>/2020-01-11-135727
  ! (bool) 0         #_default     *0   @showhouse/2020-01-11-135724
  ? wallclock/floorplan
  ? showhouse/UiDraw
```

The first line of the output shows the URI (unified resource indicator) of the resource, its type, writability, current value and the time of the last change. The next lines list the active requests and subscribers.

Lines starting with "!" show an active request. In the example above, there is a request with the ID "#_default" for a value of 0 with a low priority ("*0") and another one "#motion" for a value of 1 with a higher priority, which supersedes the default request in this situation. The "#motion" request has a time-out attribute set causing the request to be auto-removed at the given time. The "@" tags at the end of the lines identify the origin (host/time) of each request.

> *i* This example reflects the situation shortly after pressing the "M" button in the *ShowHouse* window at (simulated) daytime. The "#motion" request tries to switch on the light for 5 seconds (until 14:06:06). However, a "#daylight" request at a higher priority is active as well, which ties the value to 0, so that the light effectively remains off.

Lines starting with "?" show a subscriber. In this example, the resource is subscribed by the floor plan view of the *WallClock* and the *"UiDraw()"* function of the *ShowHouse* script, which is responsible for redrawing the ASCII art on value changes.

- Press "D" and "M" in the *ShowHouse* window to change the daylight status and simulate motion (or not) and repeat the above command in various situations to see how the respective requests change.
- Explore the directory tree of the installation! Find out, which resources are provided by which of the running *Home2L* server instances!

2.4.3. Manipulating Resources

We now manipulate the light by placing requests manually.

- Enter (*Note: The second command is the digit "one", not a lower-case "L".*)

```
home2l> c /alias/frontLight
/host/showhouse/world/light
home2l> 1
/host/showhouse/world/light[bool,wr] = 1 @2020-01-11-141039.993
  ! (bool) 1          #shell        *8   @home2l-showcase<shell:431>/2020-01-11-141039
  ! (bool) 0          #daytime      *4   @home2l-showcase<rules:359>/2020-01-11-135727
  ! (bool) 0          #_default     *0   @showhouse/2020-01-11-135724
  ? wallclock/floorplan
  ? showhouse/UiDraw
```

  and see that the light is turned on. Look at the output above: Can you identify the request you created with the "1" command?

- Make sure that night mode is set in the *ShowHouse* window (press "D" as necessary). The light remains on. Then enter

```
home2l> 0
/host/showhouse/world/light[bool,wr] = 0 @2020-01-11-141118.960
  ! (bool) 0          #shell        *8   @home2l-showcase<shell:431>/2020-01-11-141118
  ! (bool) 1          #motion       *3 -2020-01-11-141121.476   @home2l-showcase<rules↩
    ↪:359>/2020-01-11-141116
  ! (bool) 0          #_default     *0   @showhouse/2020-01-11-135724
  ? wallclock/floorplan
  ? showhouse/UiDraw
```

  This forces the light to stay off. Motion events have no effect on the light even at night, since their requests have a lower priority ("*3") than the default priority of shell requests ("*8").

- Finally, remove the manual shell request by typing

```
home2l> -
/host/showhouse/world/light[bool,wr] = 0 @2020-01-11-141118.960
  ! (bool) 0          #_default      *0    @showhouse/2020-01-11-135724
  ? wallclock/floorplan
  ? showhouse/UiDraw
```

i The commands "0", "1", and "-" are abbreviations for variants of the "r+" and "r-" commands to place and remove requests. More information is given in the online help of `home2l-shell`.

- It is possible to pass arbitrary request attributes (see Section 5.6) as command line arguments. To turn the light on in 2 seconds and off again 3 seconds later enter:

```
home2l> 1 +2000 -5000
/host/showhouse/world/light[bool,wr] = 0 @2020-01-11-141118.960
  ! (bool) 1          #shell         *8 +2020-01-11-141300.240 -2020-01-11-141303.240 ↩
    ↪ @home2l-showcase<shell:431>/2020-01-11-141258
  ! (bool) 0          #_default      *0    @showhouse/2020-01-11-135724
  ? wallclock/floorplan
  ? showhouse/UiDraw
```

  The arguments '+2000' and '-5000' add start and stop time attributes to the request so that it becomes effective in 2000ms from now until 5000ms from now.

- (Optional) Explore the directory tree and try out writable resources!

  How can you open or close the shades of the kitchen window? How can you dim or switch off the *WallClock* display?

  Finally, remove all requests again you have set ("r-" command).

2.4.4. Monitoring Resources

The shell allows to subscribe to any resources using the "s+" and "s-" commands, respectively.

- Subscribe to all resources provided by the *ShowHouse* "world" driver:

```
home2l> s+ /host/showhouse/world/*
Subscriber 'home2l-showcase<shell:431>/shell'
  /host/showhouse/world/*?
  /host/showhouse/world/bath-shades
  /host/showhouse/world/bath-window
  /host/showhouse/world/bedroom-shades
  ...
  /host/showhouse/world/simDaylight
: /host/showhouse/world/bath-shades = ?
: /host/showhouse/world/bath-window = ?
: /host/showhouse/world/bedroom-shades = ?
...
: /host/showhouse/world/simDaylight = ?
```

The first part of the output (without the lines starting with ":") shows the ID of the subscriber of the shell and lists the resources it currently subscribes to. Lines ending with "?" are *watch set* entries and are currently not associated with any existing resource.

> *i* *Watch set* entries may either be named resources or – as in this case – wildcard patterns. They allow to start and stop the subscribing and serving hosts independently from each other. You can verify this by closing both the *ShowHouse* and the *Home2L Shell*, and then repeating the above command *before* starting the *ShowHouse*.
>
> The command "s" lists the status of the shell's subscriber. Run it before and after starting the *ShowHouse*.

Lines starting with a colon (":") are events reported by the subscriber. The *Home2L* suite follows a very precise event model. Immediately after the subscription, the locally known values are reported ("unknown" in this case). However, in the background, the server(s) are contacted. By the time you have read this text, the actual values have been delivered from the server(s) to the shell.

- Press return:

```
home2l>
: /host/showhouse/world/bath-shades = 0.0% @2020-01-11-141422.985
: /host/showhouse/world/bath-shades connected
: /host/showhouse/world/bath-window = closed @2020-01-11-141422.985
: /host/showhouse/world/bath-window connected
: /host/showhouse/world/bedroom-shades = 0.0% @2020-01-11-141422.986
: /host/showhouse/world/bedroom-shades connected
...
: /host/showhouse/world/simDaylight = 1 @2020-01-11-141422.986
: /host/showhouse/world/simDaylight connected
```

Incoming events are collected in the background and displayed after each shell command.

> *i* The "... connected" events denote that the connection is established. From now on, it is guaranteed that every single value change will be reported, even if the resource value changes very quickly!

- To follow and display events immediately when received, run the "follow" command:

```
home2l> f
```

- Now make some interactions in the *ShowHouse* window (e.g. lock/unlock the door, move the person, or open and close windows). The output will be similar to this:

```
: /host/showhouse/world/motion = 1 @2020-01-11-142453.688
: /host/showhouse/world/motion = 0 @2020-01-11-142454.188
: /host/showhouse/world/lock = 1 @2020-01-11-142455.248
: /host/showhouse/world/bedroom-window = open @2020-01-11-142456.464
: /host/showhouse/world/bath-window = open @2020-01-11-142456.464
: /host/showhouse/world/lock = 0 @2020-01-11-142502.720
: /host/showhouse/world/dining-window-l = open @2020-01-11-142504.760
: /host/showhouse/world/dining-window-r = open @2020-01-11-142504.760
```

```
: /host/showhouse/world/kitchen-window = open @2020-01-11-142504.760
: /host/showhouse/world/bedroom-window = closed @2020-01-11-142504.760
: /host/showhouse/world/bath-window = tilted @2020-01-11-142504.760
```

- Press *Ctrl-C* to stop the "follow" mode. Events for subscribed resources are still reported with their correct time stamps, but only after commands have been entered.
- Quit the shell by pressing *Ctrl-D*.

2.4.5. Scripted Use and Data Logging

The *Home2L Shell* can be run non-interactively, so that certain actions can be performed from shell scripts.

- Open a command shell in the Docker container:

```
$ DOCKER bash
```

- To turn on the *ShowHouse* light for 2 seconds:

```
$ home2l shell -e "c /alias/frontLight; 1 -2000"
```

- To log all motion events, you may use a command like:

```
$ home2l shell -e "s+ /host/showhouse/world/motion; f" | tee /tmp/motion.log
```

  (Of course, the command can also be run in the background.)

- To log the outside temparature over time, run:

```
$ home2l shell -e "s+ /alias/weather/temp; f"
```

  (Again, the output may be redirected at your choice as in the previous example.)

2.5. Writing Automation Rules

Home2L automation rules are normal Python scripts. They can be tested separately from already existing rules files and later be merged into them. Section 2.5.1 gives an introduction to the Python API by means of an interactive Python session. After that, we will inspect and modify the supplied `rules-showhouse` file in Section 2.5.2.

2.5.1. Introduction to the Python Package

- Start a command shell in the Docker container (or switch to the one still open):

```
$ DOCKER bash
```

- Setup the *Home2L* environment variables and start a Python session:

```
$ . /opt/home2l/env.sh
$ python3
...
>>>
```

- Import the *Home2L* package and initialize it:

```
>>> from home2l import *
>>> Home2lInit ("myrules")
```

- Of course, online help and tab-completion is available for all commands. For example, try:

```
>>> help (RcGet)
Help on function RcGet in module home2l:

RcGet(uri, allowWait=False)
    RcGet(char const * uri, bool allowWait=False) -> CResource
    RcGet(char const * uri) -> CResource

    Lookup a resource by its URI and return a reference to it.
```

- For the next steps it is necessary to start the *Home2L* background tasks:

```
>>> Home2lStart ()
```

  This ends the *elaboration* phase and puts the *Resources* library into a ready-to-use state. Optionally, the network server would be started if configured so (it is not enabled here).

- To inspect the network environment and list the resources provided by, for example, `home2l-showhouse`, enter:

```
>>> RcHosts ()
['home2l-showcase<myrules:567>', 'server', 'showhouse', 'wallclock']
>>> RcHostResources ("showhouse")
['/host/showhouse/signal/fp_mail', '/host/showhouse/signal/fp_music', '/host/↩
    ↪showhouse/signal/fp_phone', '/host/showhouse/signal/fp_temp_bath',  ..., '/host↩
    ↪/showhouse/world/bath-shades', '/host/showhouse/world/bath-window', '/host/↩
    ↪showhouse/world/bedroom-shades', ..., '/host/showhouse/world/lock', '/host/↩
    ↪showhouse/world/motion', '/host/showhouse/world/simBadWeather', '/host/↩
    ↪showhouse/world/simDaylight']
```

(Output is abbreviated for readability.)

Inspecting Resources and Reading Their Values

- Now let us inspect some resources. The following sequence shows some commonly used commands to inspect resources and to obtain their values and states:

```
>>> rcNow = RcGet ("/local/timer/now")
>>> rcNow
(CResource) /host/home2l-showcase<myrules:567>/timer/now time ro
>>> rcNow.PrintInfo()
/host/home2l-showcase<myrules:567>/timer/now[time,ro] = 2020-01-11-150350 @2020↩
    ↪-01-11-150350.000
  (no requests)
  (no subscriptions)
>>> rcNow.ValueState()
(CRcValueState) (time) 2020-01-11-144234
>>> rcNow.Value()
1578753761000
>>> RcGet ("/local/timer/twilight/day").ValueState ()
(CRcValueState) (bool) 1
>>> RcGet ("/local/timer/twilight/day").Value ()
True
```

- Let us try to do the same with a remote resource (say, the timer of the *WallClock* instance):

```
>>> rcNow = RcGet ("/host/wallclock/timer/now")
>>> rcNow
(CResource) /host/wallclock/timer/now time ro
>>> rcNow.ValueState ()
(CRcValueState) (time) ?
>>> rcNow.Value ()
>>> str (rcNow.Value ())
'None'
```

Oops! The time is unknown? Well, this and the previous sequence is lacking one important thing: To get values of resources, they must be subscribed to. The previous example only happened to work because we used local resources.

To subscribe to the resource *rcNow* enter:

```
>>> s = RcNewSubscriber ("mysub", rcNow)
>>> s
(CRcSubscriber) home2l-showcase<myrules:567>/mysub: /host/wallclock/timer/now
>>> rcNow.PrintInfo()
/host/wallclock/timer/now[time,ro] = 2020-01-11-145439 @2020-01-11-145439.007
  (no requests)
  ? home2l-showcase<myrules:567>/mysub
  ? wallclock/floorplan
>>> rcNow.ValueState()
(CRcValueState) (time) 2020-01-11-144911
```

Repeat the last command a couple of times to see that the value is now updated regularly.

Manipulating Resources

- Resources can be manipulated by setting requests, for example, as follows:

```
>>> rcLight = RcGet ("/alias/frontLight")
>>> rcLight.SetRequest (True, "*8 -5000")
```

The last command could also be written as:

```
>>> rcLight.SetRequest (True, priority = 8, t1 = TicksOf (5000))
```

... or both together like:

```
>>> RcSetRequest ("/alias/frontLight", "1 *8 -5000")
```

Actually, the methods `CResource.SetRequest()` and `RcSetRequest()` offer a wide range of possibilities to pass arguments. Values and attributes can be passed as strings or as values by named arguments or as a mixture thereof.

Details can be found in the online documentation of these methods. To see details on passing request attributes, enter `help (RcSetRequest)` .

A Small Example Rule

We will now define a simple rule on the command line and run it. The rule monitors the current time and whenever the time changes, the value is printed out.

- Defining own drivers is only allowed before the `Home2lStart()` call. Therefore, we restart the *Home2L* package first:

```
>>> Home2lDone ()
>>> Home2lInit ("myrules")
```

- Define the rule function as follows:

```
>>> rcNow = RcGet ("/host/showhouse/timer/now")
>>>
>>> @onUpdate (rcNow)
... def OnTimerUpdate ():
...   print (str (rcNow.ValueState ()))
...
>>>
```

(Mind the indentation!)

The function *OnTime()* is prefixed with the decorator `@onUpdate`, which makes it to be executed automatically whenever the passed resource *rcNow* changes its value or state.

- Start the *Home2L* main loop:

```
>>> Home2lRun ()
```

Now, all rules (just one here) are active and executed. Interaction in the Python shell is no longer possible.

- In the *ShowHouse* window, stop the showhouse by pressing "Q". After a some time, start it again and watch the output in the *Shell* window.

```
$ home2l showhouse
```

In the output of your rule, you can see that the resource becomes unavailable and then available again.

- In the *Shell* window, press *Ctrl-C* to stop the rules process and exit Python.

2.5.2. The `rules-showhouse` Automation Script

The supplied automation script `rules-showhouse` (source tree: `tools/etc/rules-showhouse`) contains a couple of example rules, the effects of which you may have already observed in previous tutorial steps. These are rules for

- turning on the front light on motion at day time,
- closing the shades at night,
- controlling / dimming the the *WallClock* display,
- opening the roof window at night on hot days for cooling,
- determining the *presence* (use) state if the user selects the "Auto" mode.

- Changing the rules file requires root permissions. Define a macro for running commands as *root* in the Docker container:

```
$ alias DOCKER_ROOT="docker exec -ti --user=root home2l-showcase"
```

- Inspect and read the rules file:

```
$ DOCKER_ROOT nano /opt/home2l/etc/rules-showhouse
```

(Color syntax highlighting can be switched off/on by pressing *Alt-Y* in *nano*.)

Can you identify the rules mentioned above?

> *i* The rules script makes frequent use of the `ValidValue` method to obtain values. This method simplifies the handling of potentially unknown values by returning the passed default value if the respective resouce value is unknown. For example, the line
>
> ```
> if rcTempDayMaxOutside.ValidValue (0.0) < 25.0:
> ```
>
> can be read as "if the temperature is less than 25.0 or unknown, then do ...".

- In the next steps, we will modify the rules script. To facilitate editing and debugging, we will run it in the foreground as long as we work on it.

Stop the *Home2L* daemon:

```
$ DOCKER demo stop
```

(The server instance will stop, too, and the weather displays disappear. This is no problem.)

> *i* Alternatively, if you only make little changes and do not want to stop the daemon, you may also just restart the rules script by killing it:
>
> ```
> $ DOCKER_ROOT pkill rules-showhouse
> ```
>
> It will automatically be restarted by the `home2l-daemon`.

- Go to the code section controlling the front light. How is the light kept off at day time? What would you need to change to increase the light-on duration after a motion event?

Change the rule so that after a motion event the light is switched on for 3 seconds instead of the original setting.

After saving the file (*Ctrl-S* in *nano*), run the script in a new terminal window:

```
$ DOCKER bash
home2l@home2l-showcase:/opt/home2l/etc$ . /opt/home2l/env.sh
home2l@home2l-showcase:/opt/home2l/etc$ ./rules-showhouse
```

Leave this terminal open for running the script in the following steps.

- Assume you do not want to have all shades closed at night. Only those of the living room and the bedroom shall be closed, and those of the kitchen and bathroom left open.

What needs to be changed in the rule script? Try it!

- The rules for roof window cooling depends on a resource named `rcTempDayMaxOutside` which models the maximum outside temperature of the day and is used to determine whether it is a hot day or not. Since it is not guaranteed that it is a hot day when you are reading this, this resource is mapped to a local signal in this demo.

  Identify the place where this internal signal resource is defined and change its value from 20.0 (not hot) to 30.0 (hot day).

  Restart the rules script and try the roof window cooling functionality! See that the window opens during the night and the shades close during the day (press "D" in the *ShowHouse* window to switch between day and night time).

  At night time, press "R" to simulate rain. See how the window closes to avoid damage!

- The *WallClock* can switch on and off the display or dim it, depending on its use. The display state is exposed by two boolean resources `ui/standby` and `ui/active`, which allow the display state to be controlled externally by rules. So far, a rule has been active which has simply kept the display active all the time. We now replace this with a more sophisticated rule that dims the display at night and if not used.

  > ***i*** Like potentially any *Home2L* application, the *WallClock* exports a number of resources. For example, on *Android* it can export the device's brightness sensor value or Bluetooth status. In this tutorial, we will use `ui/active`, `ui/standby` for controlling the UI state and optionally `ui/mute` to mute the audio player.

  Identify the rule for keeping the *WallClock* display permanently active

  ```
  @daily("wallclock")
  def WallClockDisplay (host):
    ...
  ```

  and deactivated it by commenting out its `@daily` decorator:

  ```
  # @daily("wallclock")
  def WallClockDisplay (host):
    ...
  ```

  To make this change effective, it necessary to remove its already existing permanent requests:

  ```
  $ DOCKER home2l shell -e "c /host/wallclock/ui; r- active rules; r- standby rules"
  ```

  After some time (at most 10 seconds, see the `ui.standbyDelay` setting in `home2l.conf`), the *WallClock* UI should dim or turn off completely. Clicking on it re-activates it for some time. You may test this now, but should then wait again until the screen dims. After that, you should not click into the *WallClock* window anymore.

- Replace the `WallClockDisplay` rule with a new one, which switches the display on at day time or whenever the door is unlocked in order to remind the user to lock it over night:

```
rcLock = RcGet ("/host/showhouse/world/lock")

@onUpdate(rcDaylight, rcLock)
def WallClockDisplayNew ():
  print ("### daylight = " + str (rcDaylight.ValueState ()) + ", lock = " + str (↩
    ↪rcLock.ValueState ()))
  daylight = rcDaylight.ValidValue (False)
  doorLocked = rcLock.ValidValue (False)
  if not daylight and not doorLocked:
    rcWallClockActive.SetRequest (value = True)
  else:
    rcWallClockActive.DelRequest ()
```

(Mind the indentation!)

The first line (`@onUpdate()`), a Python *decorator*, causes the function *WallClockDisplayNew()* to be called whenever the value of any of the supplied resources changes. Inside the function body, the values of *rcDaylight* and *rcLock* are read using the `ValidValue()` method, which takes a default value as an argument and always returns a valid value, even if the actual resource value is currently unknown. Depending on these values, a request is set to switch the display active or not.

The line `print ("### ...` just serves debugging purposes and is not strictly necessary.

- Add a new rule to mute the music player for 5 seconds if motion is detected in front of the house:

```
@onEvent ("/host/showhouse/world/motion")
def MuteOnMotion (ev, rc, vs):
  if ev == rceValueStateChanged and vs.Value () == True:
    RcSetRequest ("/host/wallclock/ui/mute", "1 -5000")
```

For demonstration purposes, this example uses the `@onEvent()` decorator, which allows to track all events precisely.

- Finally, start the *Home2L* daemon again, which also starts your modified rules script in the background:

```
$ DOCKER demo start
```

2.6. *Brownies*: Integrating Do-It-Yourself Hardware

This section will introduce the *Brownies* framework and guide you to integrate an *ATtiny* microcontroller with an LED into the *ShowHouse*. Up to now, the *ShowHouse* only contained virtual hardware. We will change this and replace the outdoor previously simulated by the `home2l-showhouse` with a real light!

This section expects basic knowledge in electronics and microcontroller programming, the target audience are people planning to build their own hardware. It is recommended to read the overview section of Chapter 7 before proceeding.

Disclaimer

Building electronic circuits requires adequate knowledge in electronics. Modifying the electrical installation of a building is inherently dangerous and may result in serious damage, injury or even death if not done properly.

You expressively agree to hold the authors of the Home2L suite and of this document harmless for any property damage, personal injury and/or death, or any other loss or damage that may result from your use of the information or software provided.

This material is provided "as is". Use it wisely, it is at your own risk!

2.6.1. Prerequisites

The following additional hardware is required for this part of the tutorial:

- an *ATtiny85* microcontroller,
- basic equipment for programming the *ATtiny85* (for example, an *AVRISP mkII* programmer and *avrdude(1)*),
- an *ELV USB-i2c* interface *(or some other i2c host interfaces – see info box below)*,
- a breadboard,
- an LED,
- a 220Ω resistor.

Using a Different *i2c* Adapter

In general, this tutorial can be run with any *i2c* host adapter supported as a Linux *i2c* device (kernel module `i2c_dev`, typical device names are `/dev/i2c-*`). Just note:

1. The adapter must support *clock stretching*.
2. The tutorial assumes a power supply of 5V coming from the adapter or host. Other supply voltages between 2V and 5V are possible, too. If a supply voltage other than 5V is used, the only required change in the circuit is LED pre-resistor, which should be adapted accordingly.

2.6.2. Initializing the Microcontroller

The first step is to prepare the microcontroller. This is done with your favorite programming tool and should be done before the device is built into the circuit. This step only needs to be done once. After that, the microcontroller can be installed, and the programmer is no longer needed.

- Copy the initialization image from the Docker container to your host:

```
$ docker cp home2l-showcase:/opt/home2l/share/brownies/init.t85.elf .
```

- On your host PC, program the *ATtiny* microcontroller using your programmer and software of your choice. The initialization image `init.t85.elf` contains
  - the *maintenance* firmware,
  - initial EEPROM contents,
  - initial fuse bits (like factory defaults, but with self-programming enabled).

  Make sure that you program all of them.

  For example, with *avrdude* and an *AVRISP mkII* programmer, the command is:

```
$ avrdude -c avrisp2 -p t85 \
    -U  hfuse:w:init.t85.elf \
    -U  efuse:w:init.t85.elf \
    -U eeprom:w:init.t85.elf \
    -U  flash:w:init.t85.elf
```

> *i* This is usually the last time you need your programmer. All other programming – installing the operational firmware, configuring the device – can be done with the *Home2L Brownies* maintenance tool (`home2l-brownie2l`).

2.6.3. Building the Circuitry

The demo circuit is minimal *bus tree* with a single device node (*Brownie*). This device has an LED as an acting output.

- Build your *bus tree* circuitry on the breadboard as shown in Figure 2.4.
  - The TWI slave port signals *twi_sl_scl* and *twi_sl_sda* are connected to the *USB-i2c* adapter.
  - The LED is driven by signal *gpio0* (pin 6 / PB1).
  - Power supply is provided by the *USB-i2c* adapter.

> *i* The complete signal-to-pin assignments for the *ATtiny85* are shown in Figure 7.4.

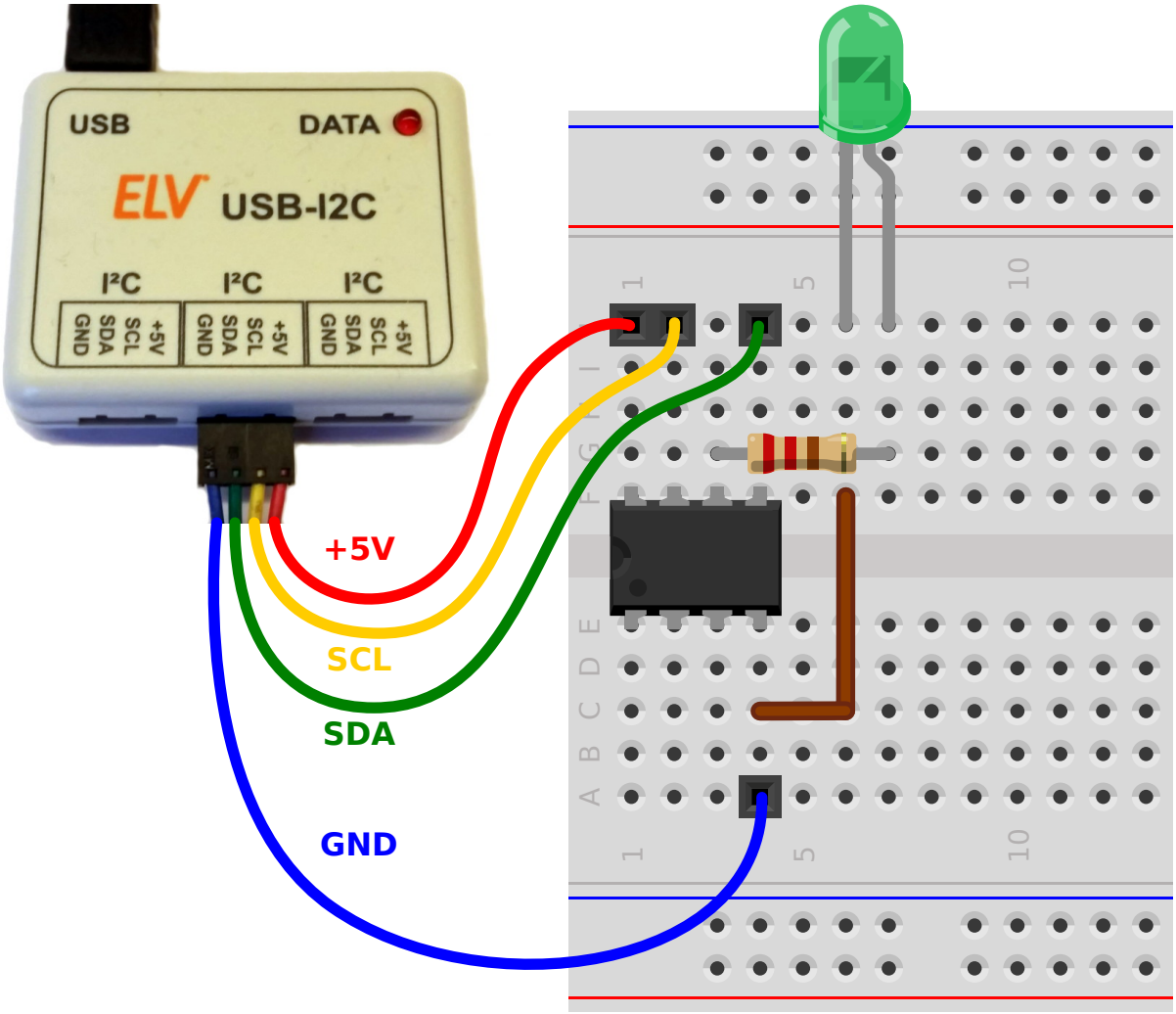


Figure 2.4.: Tutorial circuitry for a minimal *Brownie bus tree* with one *ATtiny85* microcontroller

2.6.4. Testing and Configuring the Hardware Using `home2l-brownie2l`

Now we will test the hardware step-by-step and then prepare the *Brownie* for productive use.

- Stop the running Docker container by pressing *Q* and then *Ctrl-D* in the *ShowHouse* window.
- Connect the *i2c* interface adapter to the PC.
- Start the Docker image again as described Section 2.2, but with the additional option

  ```
  ... --device /dev/ttyI2C ...
  ```

  inserted *before the last argument*. Replace *"/dev/ttyI2c"* with the actual device name of your *i2c* adapter (probably *"/dev/ttyUSBx"* in case of the ELV adapter, else like *"/dev/i2c-x"*).
- The demo system expects the *i2c* device to be named *"/dev/i2c"*. Set a symlink link accordingly:

  ```
  $ DOCKER_ROOT ln -s /dev/ttyI2C /dev/i2c
  ```

Again, replace *"/dev/ttyI2c"* with the actual device name of your *i2c* adapter.

- Start the *Brownie maintenance tool*:

```
$ DOCKER home2l brownie2l
home2l-brownie2l v0.9-233 (2020-01-26) by Gundolf Kiefer

Connected to '/dev/i2c' (ELV USB-i2c).
Read database file '/opt/home2l/etc/brownies.conf'.

brownie2l>
```

This is a check if your adapter is working from within your ShowCase environment. If it has not been detected, either your adapter is incompatible, or you have a software problem on your PC. If so: Fix it now. You may disconnect the adapter from the circuit for this.

Verify that your i2c adapter has been detected successfully!

- Run an *i2c* bus scan:

```
brownie2l> scan
007 new                     init.t85 v0.9.233*
```

This is a check if your ATtiny has been programmed and your circuitry been connected correctly. If the scan takes longer than a few seconds, there is a general problem with the bus (e.g. missing pull-ups, open wires, missing power supply).

Verify that your device has been detected as shown above! (The firmware version may differ.)

- Open the device:

```
brownie2l> open 7
007 new
        Device:   ATTiny85
        Firmware: init.t85 v0.9.233*
        Features: maintenance
        Config:
```

The output lists the features and indicates that the maintenance system is running.

- Download the operational firmware for the preconfigured *Brownie* "showgarden":

```
brownie2l> program -d showgarden

Segments in '/opt/home2l/share/brownies/ahub.t85.elf':
   FLASH  : 0a00 - 15cc (3020 bytes)
  (SRAM)  : 0060 - 00e2 (130 bytes)
  (EEPROM): 0000 - 0032 (50 bytes)

(Re-)program FLASH of device 007 with this? (y/N) y

Flashing device 007 with '/opt/home2l/share/brownies/ahub.t85.elf' ...
  0a00 - 15cc (3020 bytes) ... verifying ... OK
```

- Configure the device according to the preconfigured database (brownies.conf):

```
brownie2l> config -d showgarden

007 showgarden
  adr=008 id=showgarden fw=ahub.t85

Write back this configuration and reboot node 007? (y/N) y
Writing ID and config ... verifying ID ... verifying config ... OK
Rebooting ... OK
```

- Activate and boot the operational system:

```
brownie2l> boot -o
Switching device 008 to OPERATIONAL firmware (block 0x28, adr=0x0a00) ... Activating ↩
    ↪and rebooting ... OK
```

- Perform another (verbose) bus scan to see the device with its new identity:

```
brownie2l> scan -v
008 showgarden               ahub.t85 v0.9.233*
        Device:   ATTiny85
        Features: twihub
        GPIOs:    0
        Config:   hub_maxadr=000 hub_speed=0
```

- We will now access the device directly to switch on and off the LED. The LED is connected to GPIO #0. To switch on the LED, write a "1" into the first GPIO register:

```
brownie2l> write gpio-0 1
reg(0x02) <- 0x01
```

  To switch the LED off again, enter:

```
brownie2l> write gpio-0 0
reg(0x02) <- 0x00
```

> ***i*** Like the *Home2L Shell*, home2l-brownie2l has an exhaustive online help. Entering
>
> ```
> brownie2l> help [<command>]
> ```
>
> will display a list of all commands or help on a particular command. The complete register map including the exact name and address of the GPIO register used above is listed in the help of the *"write"* (or *"read"*) command:
>
> ```
> brownie2l> help write
> ```

- During the session, home2l-brownie2l continuously collects communication statistics. Display the statistics collected so far:

```
brownie2l> statistics

TWI Communication Statistics
============================

Sending                       | Fetching                      |
Ops         Retries  Failures | Ops        Retries  Failures |
-------------------------------------------------------------------------------
      706       508       254 |      452         0         0 | Reason
-------------------------------------------------------------------------------
                508       254 |                  0         0 |  8 No device at given ↩
    ↪address

Statistics on 'brownie2l@home2l-showcase<303>' since 2020-01-26-195435.023.
```

The errors shown here were caused by the *"scan"* commands as they tried to contact non-existing devices. The 452 (=706-254) successful operations are related to the remaining activities, mainly firmware programming and configuration.

Reset the statistics:

```
brownie2l> statistics -r
```

Then turn the LED on and off again and print the statistics again:

```
brownie2l> w gpio-0 1
reg(0x02) <- 0x01
brownie2l> w gpio-0 0
reg(0x02) <- 0x00
brownie2l> statistics

TWI Communication Statistics
============================

Sending                       | Fetching                      |
Ops         Retries  Failures | Ops        Retries  Failures |
-------------------------------------------------------------------------------
        2         0         0 |        2         0         0 | Reason
-------------------------------------------------------------------------------

Statistics on 'brownie2l@home2l-showcase<303>' since 2020-01-26-195715.520.
```

Each of the *"w"* commands caused one operation, which is visible in the output.

2.6.5. Installing the *Brownie* in the Show House

Now we will integrate the *Brownie* into the *Home2L* cluster of the demo system. Let us replace the outdoor light of the *ShowHouse* with a real light – your LED!

- Quit `home2l-brownie2l` (*Ctrl-D*) and edit the file `home2l.conf` in a text editor:

```
$ DOCKER_ROOT nano home2l.conf
```

In section "[server]", add the following line to enable the "brownies" driver for the server instance:

```
drv.brownies = 1
```

After the lines following "[brownie2l,server]", add:

```
[brownie2l]
br.link = "="
```

This directs `home2l-brownie2l` for the future to attach to the running driver instead of accessing the *i2c* device directly (otherwise, conflicts would occur if both try to access the same device).

- Edit the file `resources.conf` in a text editor:

```
$ DOCKER_ROOT nano resources.conf
```

For the front light, an alias is used. It presently points to a resource provided by the *ShowHouse* and will now be changed to point to the *Brownie's* GPIO resource connected to the LED.

Change the line defining the alias `frontLight` as follows:

```
A frontLight      server/brownies/showgarden/gpio/00
```

- To make these changes effective, all tools must be restarted. Quit the *WallClock* and the *ShowHouse* (press "Q"). Then run the following commands in the *ShowHouse* window:

```
$ home2l demo stop   # stop background services
$ home2l demo run
```

Your done!

- Interact with the *ShowHouse* or *WallClock* and see how the outdoor light is operated by your LED!

2.7. *WallClock* Gadgets: The Music Player

The *WallClock*'s music player is a Music Player Daemon (MPD) client, optimized for a home setup with multiple *WallClocks* in different rooms and multiple music machines.

- Copy some of your favorite music files into the Docker container:

  ```
  $ docker cp /your/favorite/album/ home2l-showcase:/var/opt/home2l/mpd/music/
  ```

  Replace "/your/favorite/album/" with a directory containing some music files on your computer. It should contain about $1 - 50$ *mp3* or *flac* files.

- Push the *"Music"* button in the *WallClock* window.
- In the navigation pane (right half of the screen), navigate to your favorite song. Pushing the title bar of the navigation pane will navigate up or switch between the local database and playlists.
- Select your favorite song, push the play button and enjoy!

Audio Output Troubleshooting

The integrated music player relies on a working MPD (should have been installed so far) and a working ALSA device (`/dev/snd`).

The MPD configuration file is located in `/opt/home2l/etc/mpd-showstage.conf`. It should work out-of-the-box, but may be further adapted to your preferences. By default, the audio output is directed to the default ALSA playback channel.

To play audio from the Docker container, it is important that on your host system *no other process* uses the sound card.

Such processes can be identified by:

```
# lsof 2>/dev/null | grep /dev/snd | grep -v 5000
```

To test if audio playback is working inside your container / VM, run:

```
$ speaker-test -t sine
```

To adjust the volume or unmute the device, run:

```
$ alsamixer
```

To update the music database manually after you have copied files into the music directory:

```
$ mpc -h localhost update
```

Make sure that the *MPD* instance with the *Home2L* configuration has been started properly and that the default instance is off (does not apply to the Docker container):

```
$ sudo systemctl disable mpd
$ sudo systemctl stop mpd
$ sudo systemctl restart home2l
```

2.8. *WallClock* Gadgets: The Family Calendar

The calender applet uses *remind(1)* as a backend and thus supports its syntax for specifying events.

- Start the applet by pushing "Calendar" on the main screen.
- Explore the calendar UI by navigating to different dates (e.g. your wife's next birthday or your next anniversary).
- Select an event in the right pane and modify it (e.g. change its time or text).
- Select a day in the left pane and add a new one-time appointment for Julian, for example (enter/keep your own date at the beginning):

```
2020-12-23 at 19:00 dur 2:00 MSG Meet Henry; Murphy's Pub
```

2.9. Going Further

To learn more about the capabilities of the *Home2L* suite, we suggest to look into the configuration files and source code of the tutorial.

- The supplied rules file has already been inspected in Section 2.5 (source tree: `tools/etc/rules-showhouse`):

  ```
  $ DOCKER nano /opt/home2l/etc/rules-showhouse
  ```

- Inspect and read the *ShowHouse* source file (source tree: `resources/home2l-showhouse`):

  ```
  $ DOCKER nano /opt/home2l/bin/home2l-showhouse
  ```

  It contains an example for a resource driver implemented in Python, namely for the *ShowHouse* gadgets and the keyboard input. The latter is a bit more sophisticated and involves a background thread.

  The ASCII art visualization is refreshed whenever necessary, but not unnecessarily often. This is achieved by implementing the drawing function as a rule.

- Inspect and read the main configuration file (source tree: `tools/etc/home2l.conf`):

  ```
  $ DOCKER nano /opt/home2l/etc/home2l.conf
  ```

The following major components are not (yet) covered in the tutorial:

1. Setting up an intercom system with multiple room video phones and multiple doorphones using Asterisk, `home2l-wallclock` and `home2l-doorman` instances. The author has such a system in productive use for a few years now, but it requires some effort and technical expertise (and patience) to set it up. In order to get video transmission over a PBX system and echo cancellation working properly, the *Liblinphone* library may need to be patched and re-compiled.

2.10. Alternative Ways of Running the Tutorial

2.10.1. Native Installation

Of course, the *ShowCase* demo system can be run natively on a Linux machine.

The official reference for building and installing the *Home2Ls* is the `Dockerfile`, which is both machine-readable and (at best effort) human-readable. It uses and is tested with the current stable release of *Debian* as a base system. The *Home2L* suite aims to minimize external dependencies, so that other distributions (Debian-based and others) should work with little problems.

In the `Dockerfile`, the sections entitled "Stage 1: Building" and "Stage 2: Runnable Demo Image" contain all information required for building and installing the tools, respectively.

After installing the *Home2L* suite, the demo environment as presented in Section 2.2 can be started with:

```
$ home2l demo run  # start all background services, the WallClock and home2l-showhouse
```

2.10.2. Using a VirtualBox VM

It is also possible to run the demo in a VirtualBox VM. The project provides a preconfigured VM image to facilitate this.

The following instructions have last been tested with *VirtualBox 5.2.10* and *Debian 9.6 (Stretch)*.

- Unpack the virtual machine into a new directory:

  ```
  $ mkdir home2l-tutorial
  $ cd home2l-tutorial
  $ tar xzf path/to/home2l-showcase-vbox.tar.gz
  ```

- Download and provide a Debian installation image as 'install.iso' in the same directory:

  ```
  $ wget https://cdimage.debian.org/debian-cd/current/i386/iso-cd/debian-9.6.0-i386-↩
      ↪netinst.iso
      # Adapt the path as necessary. We need an i386 "netinst" image.
      # More information can be found on the Debian download page at
      # https://cdimage.debian.org.
  $ ln -s debian-*.iso install.iso
      # or rename the file (if your OS does not support symbolic links)
  ```

  The virtual machine comes with an empty harddisk image and is configured to have the CD/DVD image *'install.iso'* inserted in its optical drive.

- Start *VirtualBox*, select "Machine → Add..." and navigate to

  ```
  home2l-tutorial/home2l-showcase-vbox/home2l-showcase.vbox
  ```

  to add the *Home2L ShowCase VM* as new virtual machine.

- Start the virtual machine:

```
$ virtualbox --startvm home2l-showcase
```

  The VM automatically boots from the Debian installation medium and runs the installer.

- During the installation, accept all default settings or leave fields empty, except for the following options:
  1. Select your country, language and keyboard layout as convenient for you.
  2. As a computer name, enter: `home2l-showcase`.
  3. Leave the root password empty (will allow *home2l* to use *sudo* for root access).
  4. As the name for the first normal user enter: `home2l`
  5. For user '*home2l*' enter a password at your choice (and do not forget it!).
- It is recommended to open this book inside the virtual machine. This will allow you to copy and paste terminal commands from this document into a terminal.
- Optional: Install the *VirtualBox Guest Additions*.
- Install and run the *Home2L* demo environment sketched in Section 2.10.1.

i

1. The *VirtualBox Guest Additions* are helpful for optimizing the screen resolution of the VM, but they are not required otherwise. If their installation fails, you can continue without them. In this case, you should move the PDF viewer to a second virtual desktop in the VM. Without it on the main desktop, a resolution of 1024x768 pixels is sufficient to run the tutorial.
2. The VM has been pre-configured to use "NAT" networking. This is the most fail-safe setting and allows the VM to share the internet connection with your host. If you want to contact the VM from your host or some other machine in your LAN, you can change the network setting to "bridged". Please consult the *VirtualBox* documentation for details.

3. Building and Installing

3.1. Overview

The official up-to-date reference for building and installing the *Home2Ls* in a normal way is the `Dockerfile`, which is both machine-readable and (hopefully) human-readable (see also Section 2.10.1).

This chapter gives some background information on the *Home2L* build system and covers some special aspects such as cross-compilation and building the Android app.

The *Home2L* suite is not a piece of software to be installed on a single computer, but instead, the *Home2Ls* are installed in a "home", which is a heterogeneous *cluster* of machines with different hardware (e.g. x86, ARM) and operating software environments (e.g. Debian, Android). For this reason, the build system supports cross-compilation and installation for multiple architectures.

3.2. Prerequisites

The requirements for building the core part of the *Home2Ls* are intentionally kept simple:

- A C/C++ compiler (compliant with the *C99* and *C++11* standards, respectively) with basic libraries (*libc*, *libstdc++*).
- *Python 3* with development packages and *SWIG* (>= 3) for the *Python API*.
- *libreadline* (optional, for the `home2l-shell`).

An exact list of packages to install on Debian or Debian-based systems can be found in the `Dockerfile`.

Building the documentation (module '*doc*') requires a couple of additional packages (*doxygen, texlive, graphviz, ...*). Most users do not have to build the documentation, since readable versions are available on the project page.

3.3. Cross-Compilation

Cross-compilation is supported by the *Home2L* build system based on the Debian cross-building capabilities for the architectures *i386*, *amd64*, and *armhf*. The development machine must have *i386* or *amd64* set as its primary architecture. The other architectures must be entered as additional architectures to the *dpkg(1)* package manager.

To cross-build *armhf* binaries on an *i386* or *amd64* machine, as of Debian 10 ("Buster"), the following packages must be installed:

```
crossbuild-essential-armhf g++-8-multilib
```

For all desired target architectures, the respective development packages mentioned above must be installed.

3.4. External Libraries

Some applications require additional external libraries, sometimes depending on the options they are compiled with, as indicated in Figure 1.1. The folder `external/` in the source tree contains hints on how to obtain, build and setup the respective libraries for different platforms. Please note, that this folder is distributed "as is" without any warranty for correctness and completeness.

Files to watch for are:

`prebuild.sh`: A build script with comments on how to obtain the sources and hints for building.

`Debian.mk`: A Debian/Linux makefile fragment.

`Android.mk`: An Android NDK makefile fragment.

3.5. Building and Installing on the Master Machine

To build and install the suite, do the following:

1. View the build options by running the following command in the main source directory to view the build options:

   ```
   $ make help
   ```

2. Check and, if necessary, adapt the compiler and build settings in file *Setup.mk*.

3. Build:

   ```
   $ make <options>
   ```

4. Install the *blob* to *$HOME2L_ROOT* (default: */opt/home2l*):

```
$ make <options> install
```

This will install the so-called *Home2L blob* on your computer. It contains all files for all architectures together with a sample configuration (in *$HOME2L_ROOT/etc/*). This *blob* can now be simply copied to all machines of your cluster. The `home2l-rollout` tool can automate that for software or configuration updates. To use the *Home2Ls*, some additional things have to be set up. This is explained in the following sections.

3.6. Setting Up Users and Permissions

3.6.1. Users and Groups

The following users are involved and must exist on each machine:

- User *'home2l'* with primary group *'home2l'*: Under this UID, all *Home2L* background processes are executed. This user should get all permission required for its background or end-user tasks, but no more than that. In particular, *'home2l'* should not be allowed to modify configuration files.
- User *'root'*: The super user.
- The user account of the administrating user - we call her *'myadmin'* here.

The user *'home2l'* must have *'bash'* as its login shell and the following line in its *.bashrc* file to have all *'HOME2L_*'* environment variables set when required:

```
$ source $(home2l -e)
```

3.6.2. Automatic *ssh* Logins and *sudo* Rules for Cluster Administration

For central cluster adminstration using `home2l-rollout`, the following *sudo* rules are required on each machine of the cluster:

- for *'myadmin'*: run `home2l-install` as *'root'*
- for *'myadmin'*: run *adb(1)* as *'home2l'* (only on machines hosting Android devices)
- for *'home2l'*: run `home2l-sudo` as *'root'* (optional)

The following *ssh* logins must be possible without a password in the cluster ("*master*" is the master machine):

- from *'myadmin@master'* to any other machine as user *'myadmin'*,
- from *'root'* at any non-master host to *'home2l@master'*.

3.6.3. File Permissions

The file permissions in the installation directory (including *etc/*) are maintained by the tools `home2l-install` and `home2l-rollout` as follows:

- `home2l-install` sets the ownership to *'root:home2l'*. Permissions are preserved from the master, *'make install'* sets them to 644 for files and 755 for directories.
- The folder *etc/secrets* is meant for storing sensitive data only readable by members of the group *'home2l'*. The permissions are set to 640 for files and 750 for directories. Hence, only *'root'* or users of group *'home2l'* can read them, and only *'root'* can modify them. Others have no access.

3.7. Adding a New Machine to the Cluster

To install the *Home2L* suite on the first computer, follow the building and installation steps described Section 3.5 to install a *Home2L blob* on your *master computer*. To complete the installation, run (assuming that the blob is installed in */opt/home2l*):

```
$ sudo /opt/home2l/bin/home2l-install -i
```

The tool explains itself what it is doing.

To prepare a new additional (Linux) machine and add it to the cluster, the installation blob can be cloned from an existing (typically the *master*) machine.

1. On the *master*: Edit the configuration files `rollout.conf`, `install.conf` and `init.conf` to reflect the new setup with the new machine.
2. On the new machine: Setup users and their rights as described in Section 3.6.
3. On the new machine: Replicate the blob from the master by running a command like

   ```
   $ sudo rsync -va --perms --chown=root:home2l home2l@master:/opt/home2l/ /opt/↩
       ↪home2l
   ```

4. On the new machine, run:

   ```
   $ sudo /opt/home2l/bin/home2l-install -i
   ```

5. On the master, run

   ```
   $ home2l-rollout
   ```

   and check, if the new machine is listed. Press 'y' ("yes") to run the rollout procedure and see if updating the new machine works without errors.

From now on, software and configuration updates can be rolled out by the tool `home2l-rollout` from the master.

3.8. Installing the *WallClock* on Android

Compiling and installing the *Android* app requires some knowledge in *Android* development. The maintenance tool `home2l-adb` is still experimental.

To integrate an Android device running the *WallClock* into the *Home2L* cluster, the following steps have to be done:

1. Add a new entry for your Android device to `androidb.conf` with the following columns:
   a) Host name of your Android device,
   b) ADB host (= the Linux machine to which the device is connected, e.g. to install updates via ADB),
   c) the ADB device ID, to be obtained by: `$ adb devices -l`,
   d) (optional) the local port for port forwarding (just in case you plan to use *OpenVPN* for a wired/radioless network setup; otherwise, ignore this column).
2. Install the *WallClock* Android app.

   ```
   $ home2l adb <your_android_host> x-install-apk
   ```

   Start the app once. It should start up with a demo setup and is not yet integrated in your cluster. Then rollout your cluster configuration:

   ```
   $ home2l adb <your_android_host> x-install-etc
   ```

3. Some functionality (presently the *WallClock Calendar*) requires that the app runs commands on a Linux host via *SSH* as user *home2l*. To set this up:
   a) Generate an SSH identity without and store it as `'etc/secrets/ssh/<your_android_host>[.pub]'`.
   b) Make sure that an SSH daemon is installed on the Linux host and the user *home2l* exists.
   c) Test the connection from an Android shell:

      ```
      $ home2l adb <your_android_host> x-ssh <your_linux_host>
      ```

      This should open a shell on your Linux machine. Make sure that you have proper access to e.g. your calendar files from this shell, then exit the shell.
   d) Follow the instructions shown to enable logins without a password.

If the *WallClock* app fails to start due to some configuration problems, additional information for diagnosis can be found in the Android log system. This can be viewed using *adb(1)*:

```
$ adb logcat -v time home2l:D SDL:V *:E *:W *:I
```

4. Administration and Configuration

A typical *Home2L* installation is distributed over multiple computers (*machines*), on each of which one ore multiple *Home2l instances* may be running. There is no central server, all *Home2L instances* are equal in rank.

4.1. Terminology

In this document, the following terminology is used:

A *machine* is a physical computer.

A *Home2L instance* is a running program (process) using the *Home2L* library, such as `home2l-wallclock`, `home2l-shell`, or a rules script. Each instance is identified by its ***instance name*** (or *instance ID*), which is usually the name of its executable (without a leading "home2l-"), but may be set to a different value by the respective application (see `sys.instanceName`). The instance name must be unique in the cluster.

A *Home2L cluster* (or *Home2L network*) is a set of machines running *Home2L* instances that interact with each other and make up the *Home2L* installation.

The *master* machine is the computer from which a cluster is managed, typically the desktop PC of the adminstrator. From this machine, software and configuration updates are rolled out.

A *Home2L server* is an instance exporting *Home2L* resources (see Chapter 5).

A *Home2L client* is an instance accessing resources (i.e. any process incorporating the *Home2L Resources* library).

A *Home2L host* a *Home2L* server or client, in other words: an instance communicating in a *Home2L cluster*. It is identified by the ***host ID*** as declared in `resources.conf`.

i

The term *'host'* is frequently used as a synonym for a computer or machine. However, its meaning may as well differ slightly. For example,

a) A *host name* may refer to an IP address in a network, and as such identify an interface of a computer, not the computer itself, which may have multiple interfaces.

b) The term *host* may refer to the operating system running on physical hardware as opposed to the guest operating sytem running in a virtual machine on the same hardware.

In the *Home2L* context, a *host* actually refers to a *Home2L* instance running on a machine. Often, there is only one instance running per machine, so that its host ID is equivalent to the machine name (aka "hostname"). However, there may as well be multiple servers with different *host IDs*.

To avoid ambiguities, this book avoids to use the term *host* for anything other than a *Home2L host*, even if it is common to call a machine "host", speaking of a "hostname" as the name identifying a machine etc. .

4.2. Maintenance Tools

The common tools for maintaining *Home2L* installations are:

- **`home2l-shell`:** The command line interface and "swiss army knife" to access and inspect the *Home2L* resources. Details can be found in Section 5.7.
- **`home2l-rollout`:** Tool to distribute configuration changes or software updates from the master to the other machines.
- **`home2l-install`:** Internal tool for performing various installation tasks on the local machine. This tool is usually not called manually, but indirectly by `home2l-rollout`.
- **`home2l-sudo`:** (optional) Container to allow the *home2l* user to perform a limited set tasks with *root* privileges (e.g. to restart certain system services if they fail to ensure long-term availability). The use of this tool is optional, to use it, the file `/etc/sudoers` has to be set up such that user *home2l* can run it as *root*. The allowed activities are encoded in the tool itself. Please note, that the tool is presently implemented as a shell script and therefore prone to potential security holes.
- **`home2l-adb`:** (optional) Wrapper for the *Android Debug Bridge (ADB)*, allows to maintain Android machines connected by USB cable to Linux machines (requires `androidb.conf` to be set up).

Each of the tools implements a `-h` (help) option, which gives up-to-date usage information.

Tools can generally be invoked in one of two ways:

a) Calling the tool directly after setting the *Home2L* environment settings, for example:

```
$ source /opt/home2l/env.sh     # This line may be put into .bashrc .
$ home2l-shell
```

b) Using the general invoker without source'ing *$HOME2L_ROOT/env.sh* first, for example:

```
$ home2l shell
```

The script named *home2l* sets the environment variables and calls the tool passed as arguments. To make this work, a symbolic link to *$HOME2L_ROOT/bin/home2l* must exist somewhere in the search path (e.g. in */usr/local/bin*)

4.3. Configuration Files

A complete *Home2L cluster* is configured by a single, common set of configuration files, which are stored on each machine of the cluster in a synchronized way. Whenever changes are necessary, the administrator edits the configuration on the *master* and replicates changes to the other machines by calling `home2l-rollout`.

> ***i*** This strategy of replicated configuration files is motivated by nature: Each individual cell of a living organism on earth contains an identical copy of the complete DNA of the respective species. In nature, this appears to be a good strategy for quite some million years of evolution now. It cannot be too bad ...

The commonly relevant configuration files are:

- **`home2l.conf`:** Main configuration file (see Section 4.4).
- **`resources.conf`:** Information for the *Resources* library (see Chapter 5).
- **`brownies.conf`:** Declaration of a *Brownie* bus tree (see Chapter 7).
- **`rollout.conf`:** Declaration of machines in the cluster.
- **`install.conf`:** Installation options of machines in the cluster.
- **`init.conf`:** Configuration for the *Home2L* init script and daemon.
- **`androidb.conf`:** (optional) Declaration of all Android devices, if `home2l-adb` is used to manage them from the master.

The syntax and contents of the main configuration file and the *Resources* configuration are explained in Sections 4.4 and 5.4, respectively.

For the other files, explanations can be found as comments inside the files themselves. A set of sample configuration files is automatically installed with a new installation.

4.4. Main Configuration File: `home2l.conf`

4.4.1. Overview

The main configuration file is expected to be *$HOME2L_ROOT/etc/home2l.conf*. An alternative file can be specified by the `HOME2L_CONFIG` environment setting or the *"-c"* command line option of most tools.

All *Home2Ls* and all applications linked against the *Home2L* library have access to parameters defined there via the `EnvGet()` and `EnvGet<type>()` API calls.

The syntax is based on that of INI files. The file contains a set of lines of parameter assigments in the format:

```
<key> = <value> [ ( ";" | "#" ) <comment> ]
```

The characters ';' and '#' start a comment, everything following them is ignored up to the end of the line. A value may optionally be quoted by single (') or double (") quotes. From unquoted values, leading and trailing whitespaces are stripped. A backslash (\) can be used to escape a single character.

Examples:

```
example.someInt = 17            # some integer value
example.string1 = Hello world!  # a string with 12 characters
example.string2 = "  space  "   # a string with 9 characters
example.string3 = ' ";" '       # a string containing a semicolon and double quotes
example.string4 = ' \"\' \" '   # a string containing both types of quotes
example.bool1 = 0               # Boolean value, equivalent to '-', 'F', 'f', ...
                                # ... 'false', 'False', or 'FALSE'
example.bool2 = true            # Boolean value, equivalent to '1', '+', 'T', 't', ...
                                # ... 'True', or 'TRUE'
```

Keys are case-sensitive. Additional parameter settings can be specified by the `HOME2L_EXTRA` environment variable (which precedes over settings in the configuration file) and as command line parameters for most tools (which would precede over the configuration file or `HOME2L_EXTRA` settings).

For example, the following command runs the *Home2L Server* application with additional debug output:

```
$ home2l server debug=1
```

The contents of the main configuration file can be split into multiple files. The following assignment has a special meaning and acts as an *include* directive. Multiple and nested includes are possible.

```
include = <path relative to $HOME2L_ROOT/etc>
```

The available set of keys is tool-dependent. Unknown keys are ignored silently.

The ommonly relevant parameters are explained in Section 4.7. Most tools have tool-specific parameters. They are listed and explained with the documentation of the respective tool.

4.4.2. Section Specifiers

It is possible to make assignments specific only to certain *Home2L* instances, machines, architectures, or groups or combinations of those. This is done by defining sections with specific headers:

```
[ <section specification> ]
```

The section specification defines to which instances the following parameter assignments apply. It can be a single *tag* or an expression with multiple *tags* and logical operations.

The following *tags* are defined on an application startup.

- The machine's host name.
- The *Home2L* instance name (usually the name of the tool without the leading *'home2l-'*, user-defined scripts may define their own arbitrary instance names).
- The operating software environment (presently `Debian` or `Android`).

> *i* The *Home2L host* ID is *not* available as a tag, since it may indirectly depend on settings of the `home2l.conf` file, and a cyclic dependency may occur.

Tags are case-sensitive. The user must take care that there are no name conflicts between machine names and instance names (both should be all lower-case). Section specifications may contain wildcards (`*` or `?`). In particular, the heading `[*]` starts a general section that applies to any instance.

The following logical operators are supported (ordered by decreasing precedence):

- Logical *NOT*: `!`
- Logical *AND*: `&`, `@`
- Logical *OR*: `,`, `+`

Examples:

```
[eniac,z3]      # Following settings apply to machines 'eniac' and 'z3'.

[wallclock@z3]  # Select the Home2L WallClock instance on 'z3' only.

[!shell]        # Select any instance except the Home2L Shell.

[*]             # Following settings apply to all instances.
```

4.5. Managing Background Services

The *Home2L Daemon* (`home2l-daemon`) can be used to start *Home2L*-related services at boot time, for example

- rules scripts,
- resource servers,
- doorman tasks,
- or any other user-specified shell script or command.

Services are specified by a `daemon.run.<script>` setting. They are kept alive by the daemon, crashed services are restarted automatically.

To enable the *Home2L Daemon* on a particular host, the file *$HOME2L_ROOT/install/initd-home2l* must be copied to */etc/init.d/* as *home2l*. By default, the init script expects the *Home2L* installation blob to reside in */opt/home2l*. To select a different path, create a file */etc/default/home2l* with a line:

```
HOME2L_ENV=<your path to the 'env.sh' file>
```

The correct setting for a running installation can be obtained by:

```
home2l -e
```

4.6. A Note on Security

Security is a serious aspect in networking and particularly smart home applications today. The security concept of the *Home2L* suite follows a) the Unix philosophy "Do one thing, and do this well" and b) the general philosophy to keep security-related things as simple and transparent as possible.

For this reason, the *Home2Ls* do *not* implement any encryption or authentication mechanisms themselves, but are designed to rely on and colaborate with existing tools and mechanisms.

The *Home2L* Security Rule

The *Home2L Resources* library and its networking operations assume to run on *trusted* computers in a *trusted* network. This is a requirement, and it is up to the user/administrator to provide such an environment, for example, a trusted LAN with trusted computers only.

If there are untrusted machines in a private home (who wants that?), a trusted network for the *Home2L* cluster must be set up, for example, using VLAN techniques or SSH tunnels.

The following configuration options are related to network security. This set is intentionally kept short and simple:

- `rc.enableServer`: If not set, *Home2L* will not listen on any networking port.
- `rc.serveInterface`: This option allows to restrict incoming connections to a certain physical network interface. If set to "local", only connections via the local interface (127.0.0.1) are accepted. This allows a setup where all peers are connected via secured SSH tunnels.
- `rc.network`: Declaration of the local network. Connection attempts from outside are dropped.

Violations against rules imposed by those settings such as connection attempts from outside the trusted network are logged with the special tag "SECURITY" to facilitate intrusion detection.

> ***i*** The *Home2L* security rule is inspired and intended to correlate with how private buildings are usually protected in the real world: All parts of some well-definable *outer hull* (e.g. the outer walls) – the main door, windows, side entrances – are secured and lockable, wheras the interior is treated as a trusted area. The main door is usually strong and lockable, whereas room doors inside a private home are usually left unlocked.

4.7. List of Common Configuration Parameters

4.7.1. Parameters of Domain `debug`

`debug` (int) [= 0] `common/base.C:272`

Level of debug output.

A value of 0 disables debug output messages. Higher values increase the verbosity of the debug output.

`debug.enableCoreDump` (bool) [= false] `common/env.C:46`

Enable generation of a core dump using the setrlimit() system call (without size limit).

4.7.2. Parameters of Domain `home2l`

`home2l.config` (string) [= "etc/home2l.conf"] `common/env.C:56`

Main configuration file (read-only).

`home2l.version` (string) `common/env.C:59`

Version of the Home2L suite (read-only).

home2l.buildDate (string) common/env.C:62

Build date of the Home2L suite (read-only).

home2l.os (string) [= ANDROID ? "Android" : BUILD_OS] common/env.C:68

Operation software environment (Debian / Android) (read-only).

This setting is determined by the build process.

home2l.arch (string) [= ANDROID ? NULL : BUILD_ARCH] common/env.C:73

Processor architecture (i386 / amd64 / armhf / <undefined>) (read-only).

This setting is generated during the build process and taken from the processor architecture reported by 'dpkg –print-architecture' in Debian.

4.7.3. Parameters of Domain sys

sys.cmd.<name> (string) [= NULL)] common/base.C:2456

Predefine a shell command.

For security reasons, shell commands executed remotely are never transferred over the network and then executed directly. Instead, a server can only execute commands predefined on the server side. This group of settings serves for pre-defining commands executed by a restricted shell.

sys.syslog (bool) [= false] common/env.C:85

Set to write all messages to syslog.

sys.machineName (string) common/env.C:90

System host name (read-only).

sys.execPathName (string) common/env.C:93

Full path name of the executable (read-only).

sys.execName (string) common/env.C:96

File name of the executable without path (read-only).

sys.pid (int) [= 0] common/env.C:99

System process ID (PID) (read-only).

sys.instanceName (string) common/env.C:104

Instance name (read-only).

The instance name should uniquely identify the running process. There is no technical mechanism to enforce uniqueness. Hence, it is up to the administrator take care of that.

The instance name can be set by the tool programmatically, or in some tools with the '-x' command line option. By default, the instance name is set to the name of the executable without an eventually leading "home2l-".

sys.droidId (string) [= "000"] common/env.C:115

Droid ID.

This is the 3-digit number displayed on the wall clocks to indicate the serial number of the device. If the host name ends with three digits, the droid ID is automatically taken from that.

sys.rootDir (char*) common/env.C:124

Home2L installation root directory (HOME2L_ROOT).

sys.etcDir (string) common/env.C:127

Root directory for configuration data (HOME2L_ROOT/etc).

This setting can only be modified via the command line or the HOME2L_CONFIG environment variable, but *not* in a config file (guess why!).

The path must be an absolute path.

`sys.varDir` (string) [= "var"] `common/env.C:135`

Root directory for variable data (HOME2L_ROOT/var).

The path must be an absolute path.

`sys.tmpDir` (string) [= "tmp"] `common/env.C:140`

Root directory for temporary data (HOME2L_ROOT/tmp).

The path must be an absolute path.

`sys.locale` (string) `common/env.C:147`

Define the locale for end-user applications in the 'll_CC' format (e.g. "de_DE").

This setting defines the message language and formats of end-user applications. Only end user applications (presently WallClock) are translated, command line tools for administrators expect English language skills.

4.7.4. Parameters of Domain `net`

`net.resolve.<alias>` (string) `common/env.C:160`

Define a manual network host resolution.

When a network host is contacted, this environment setting is consulted by the client before any system-wide name resolution is started. This can be used, for example, with SSH tunnels to map the real target host to something like 'localhost:1234'.

Another use case is just a hostname resolution independent of a DNS service or '/etc/hosts' file, which can be useful on Android client devices, for example.

4.7.5. Parameters of Domain `phone`

`phone.ringbackFile` (path) [= "share/sounds/ringback.wav"] `common/phone.C:31`

Ringback file (Linphone backend only).

This is the sound to be played to the caller while ringing.

`phone.playFile` (path) `common/phone.C:37`

Phone play file (Linphone backend only).

This is the background music played to a caller during transfer. (may be removed in the future since PBX systems like ASTERISK already provide this functionality)

4.7.6. Parameters of Domain `daemon`

`daemon.minRunTime` (int) [= 3000] `tools/home2l-daemon.C:33`

Minimum run time below which a process is restarted only with a delay.

`daemon.retryWait` (int) [= 60000] `tools/home2l-daemon.C:36`

Restart wait time if a processes crashed quickly.

`daemon.pidFile` (string) `tools/home2l-daemon.C:39`

PID file for use with 'start-stop-daemon'.

`daemon.run.<script>` (string) `tools/home2l-daemon.C:42`

Define a script to be started and controlled by the daemon.

5. *Home2L Resources*

5.1. Overview

The central component of the *Home2L* suite is the *Resources* software library. It manages, provides access to and publishes what is referred to as *resources*. A ***resource*** can be anything that can provide or take data that may change over time. Examples for resources are:

- physical sensors (temperatur sensors, motion detectors, ...),
- physical actors (window shades, room lights, ...),
- computers (that may be woken up or shut down remotely),
- software services,
- run-time changable options for some software (for example, the *Home2L WallClock* can be requested to switch on/off or dimm the screen),
- run-time status reports of some software (for example, the *Home2L WallClock* can report the display brightness on Android tablets).

The resources are managed in a completely distributed way. There is no central server and, consequently, no single point of failure. Running program instances linked against the *Home2L Resources* library communicate with each other on a peer-to-peer basis. Since they all share the same configuration, each peer is aware of the members of its cluster.

To deal with resources, operations are required to retrieve information (**values and states**) from sensing resources and to manipulate acting resources. However, there is a lot of inherent parallelism and asynchronous behavior in the system: Sensors (resources) may change their values any time (and sometimes very quickly), users behave asynchronously, and machines in a network behave asynchronously.

For reading out resources, **a subscription and event model** is implemented. If an application subscribes to a resource, each value or state change will be delivered instantaneously to the application by means of events (see Section 5.5). *Home2L* is very accurate here - all events reported by a driver can be delivered without any loss to any host subscribing to them. As an extreme example: If there is a mechanical switch which is not properly debounced, any host in the cluster is able to count the exact number of bounces!

As for the manipulation of resources, special care has to be taken about concurrency. For example, a user may push a button to open the window shades, and one second later, an automatic script may request the shades to close. What should happen now? *Home2L Resources* provides a resolution mechanism to deal with such situations properly. Any instance which intends to manipulate a

resource submits a **request** with the intended value and optionally some parameters (e.g. time interval, priority), and the *Home2L Resources* library resolves them properly in the case of concurrency (see Section 5.6).

5.2. Concepts and Terminology

Figure 5.1 shows the terminology and interaction between multiple instances. The protagonists are *applications*, *drivers*, and the *Home2L Resources* library. *Applications* can manipulate any resource of the cluster by placing ***requests*** for value changes and read out resources by ***subscribing*** to them. *Drivers* make resources available via a simple API: They ***report*** new values from their devices whenever adequate, and for acting resources, they implement a ***drive*** function to apply new values and let the associated actions happen. The primary task of a driver is to physically access its devices or services.

The *Home2L Resources* library is the glue between applications and drivers. It takes care of

- request evaluation and resolution,
- event propagation to subscribers,
- network transparency and communication with other hosts,
- organizing all resources in a unified namespace (the directory),
- handling network errors and resource failures.

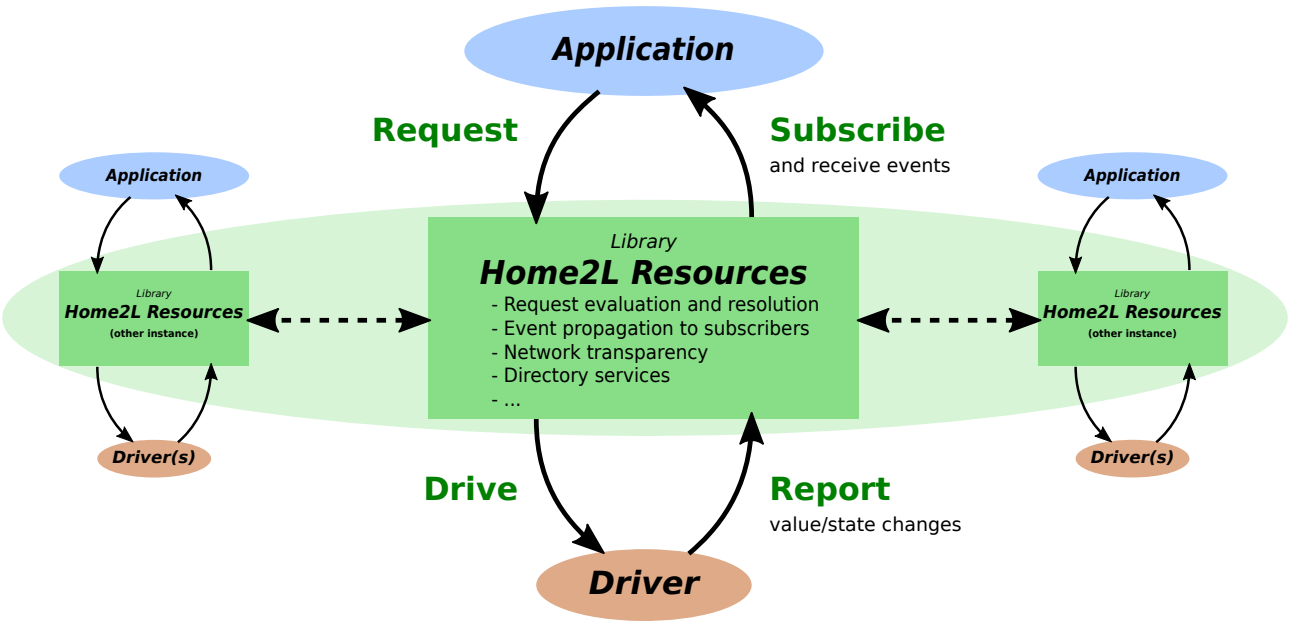


Figure 5.1.: *Home2L Resources*: Terminology

Any application linked against the *Resources* library can provide drivers or load any of the existing drivers (see Section 5.8). The same holds for automation rules scripts (see Chapter 6), which may themselves implement drivers or execute external ones.

In order to "just" load driver(s) and export their resources to the cluster, the tool `home2l-server` can be used. This is basically an empty application without any functionality besides operating the *Resources* library.

Applications can participate in the cluster either with or without running a ***server***. The server is required to provide resources to other instances. Typically, the server is enabled in applications hosting drivers, but disabled in rules scripts. User applications can enable the server, depending on whether they have resources to share. The *Home2L Shell* (`home2l-shell`) does not run a server by default, but it can be configured to run it and to load drivers, for example, in order to develop and test drivers.

i

Checklist: Enabling a *Home2L* Server

Each application linked against the *Resources* library has the ability to host drivers and to run a server in the background.

To enable the local server, the following conditions must be met:

1. The instance must be declared as a host in `resources.conf`.
2. The application must enable it when calling `RcInit()` (C/C++) or `Home2lInit()` (Python).
3. The configuration option `rc.enableServer` must be "true".

For security reasons, the server is not started (and no listening network port is opened) if any of these conditions is not met.

Client functionality is always available.

Hosting drivers does not strictly require that the server is enabled, but the resources will then only be available locally.

5.3. Resources, Values and States

5.3.1. Resources

A *resource* object has the following attributes:

- a unique resource identifier (URI),
- a "writeable" flag,
- a typed value and state,
- a list of listening subscribers,
- a list of pending requests.

Resources are organized in a namespace resembling a directory tree with the following structure:

```
/host/                          # Host domain: All resources, ordered by host/driver
    <host 1>/
        <driver 1.1>/
            <resource 1.1.1>    # Resources may be arranged in sub-trees of arbitrary ...
            <resource 1.1.2>    # ... depth by the driver.
            ...
        <driver 1.2>/
            <resource 1.2.1>
            ...
    <host 2>/
    ...
/alias/                         # Alias domain
    <alias 1>                   #   Aliases may be arranged in sub-trees and may point ...
    <alias 2>                   #   ... anywhere into the host or alias domain ...
    ...                         #   ... (like symlinks).
/local/                         # Automatic alias for /host/<local host>/...
    <driver L.1>
        <resource L.1.2>
        ...
```

Except for the */local* domain (a shortcut for a process to access its own resources), any host in the cluster has the same view on the resource tree.

5.3.2. Values

Resource values are typed. Possible types are:

a) Basic types

- *bool:* Boolean
- *int:* Integer
- *float:* Floating point
- *string:* Text string
- *time:* Time value

b) Special types

- *trigger:* Triggerable events (example: *timer/daily*)
- *mutex:* Allows to implement mutual exclusion in a *Home2L* cluster (experimental)

c) Physical (or unit) types

- *percent ::= <float> %*
- *temp ::= <float> °C*

d) Enumeration types

- *use ::= { day, night, away, vacation }*

- *window ::= { closed, open, tilted, openOrTilted }*
- *phone ::= { idle, ringing, call }*

The set of unit types and enumeration types may be extended in the future. The following command prints an up-to-date list of all types for the current installation:

```
$ home2l shell -e types
```

5.3.3. States

A resource can assume one of the following states:

- *valid* (`rcsValid`): The value is known and valid.
- *busy* (`rcsBusy`): The value is known, but the resource is busy.
- *unknown* (`rcsUnknown`): The value is unknown.

The *busy* state may indicate that an actor-like resource is anticipating some new value, but has not yet reached it. For example, if window shades are requested to close (value 0), the state/value may be *busy/0* (also written as "*!0*") while the shades are moving down and then change to *valid/0* when the shades are actually down. Or, if a computer is requested to start (value *true*), the driver may report a state/value of *busy/true* while the machine is booting and then later switch to *valid/true* as soon as it is ready to use.

The *unknown* state may be reported for a variety of reasons, for example:

- The driver may report it because its device does not deliver valid data.
- The network connection to the serving host was lost.
- The resource does not exist or the host is temporatily down.
- The resource is not subscribed to or not yet initialized.

It is important to note that application developers do not have to take extra actions for error checking. Whenever anything happens that may make the actual state unsure, the *Resources* library will turn the state to *unknown* and report this to the subscribers.

The observed state of a resource is not necessarily the same on all hosts. For example, a resource may be *valid* on the serving host, but *unknown* on some other host due to a network problem.

5.4. Configuration

The configuration file `resources.conf` defines hosts, aliases, and optionally signals for the *Home2L* cluster.

A *host* in this context is a *Home2L instance* running on a machine. In many cases, if there is only one instance serving resources, this instance can and should then be identified by its Unix host name in the network and listen on the default *Home2L* port (see below). If there are multiple serving instances running on the same machine, listening ports must be assigned manually for all but one of them.

The default port is defined by the following line:

```
P <port>
```

Hosts are declared as follows:

```
H <host ID> [ <hostspec> ]   : <hostspec> ::= [<instance>@]<net host>[:<port>]
```

<host ID> is the host ID as visible by other instances. If no *<hostspec>* is given, *<host ID>* is equivalent to the network host name, and the default port is used.

<net host>[:<port>] is the network host name and port the server is listening on.

<instance> is the instance name of the respective server. It is only needed if multiple servers run on the same machine.

Examples:

```
H eniac   # Single host on machine 'eniac', using the default server port.
H z3-server    server@z3
          # The host ID 'z3-server' refers to a 'home2l-server' process on machine 'z3',
          # and the server listens on the default port.
H z3-doorman   doorman@z3:4701
          # The host ID 'z3-doorman' refers to a 'home2l-doorman' process on
          # machine 'z3', and the server listens on port 4701.
```

Servers/hosts are also implicitly declared by the aliases and variables, which is equivalent to an explicit declaration without *<hostspec>*.

Aliases are defined with the following syntax:

```
A <aliasName> <destPath>
```

<destPath> may be relativ to */host* or absolute. Examples:

```
A tempOutside           /host/gatewayhost/weather/tempOutside
A door/backlight        doorkeeper/gpio/doorBacklight
```

These make the current outside temperature (presently delivered via internet by host *gatewayhost*) accessible as */alias/tempOutside* and the backlight of the door button as */alias/door/backlight*.

Signals are resources that do not drive any hardware or software, but simply report back all values driven to them (similar to loopback devices in Unix). They are useful for intermediate values (for example, the logical "and" of a motion sensor and darkness sensor to indicate whether a light should be switched on) or for testing purposes (for example, to temporarily replace a real device by the signal resource, which can be manipulated manually). They are defined as follows:

```
S <host> <name> <type> [<default value>]
```

Signals are handled by the `home2l-drv-signal` driver, all signals are read- and writable. Their *URI* is */host/<host>/signal/<name>*. Examples:

```
S turing testBool  bool
S turing testInt   int   7
S turing testFloat float 3.14
```

5.5. Subscriptions

Subscriptions are handled by the class `CRcSubscriber` (see C/C++ API). A subscriber object can subscribe to any number of resources, eventually selected by wildcards. Resources do not need to exist yet at the time they are subscribed to.

Subscriber events can be received in multiple ways (see class `CRcEventProcessor`):

- asynchronously by providing a callback function,
- synchronously and blocking (`CRcEventProcessor::WaitEvent ()`),
- synchronously and non-blocking (`CRcEventProcessor::PollEvent ()`).

The following types of events may be delivered (see class `CRcEvent`):

- `rceValueStateChanged`: The resource has changed its value.
- `rceDisconnected`: The connection to the (remote) resource has been lost.
- `rceConnected`: The connection to the (remote) resource has been (re-)established.

The events `rceDisconnected` and `rceConnected` are relevant if each value/state change must be captured and not just to get the most up-to-date value. These events indicate whether a gapless event sequence is guaranteed or not.

For those missing "just a normal read" operation: As long as a resource is subscribed to by some subsciber, a call to `CResource::GetValueState()` and its relatives always returns the most up-to-date value and state.

With the Python API, subscribers are used internally whenever the decorators `@onEvent(<resources>)` or `@onUpdate(<resources>)` are used. Functions decorated with the

former are executed with each event on one of its resources. The latter is more efficient since it may drop value/state changes if they are already outdated at the time they are fetched.

5.6. Requests

In order to properly handle concurrent manipulating accesses to resources, a *request resolution* mechanism is implemented. It loosely resembles the concept of *resolution functions* in hardware description languages (VHDL, Verilog and friends). Applications never "write" to a resource directly (this may cause conflicts), they place *requests* instead.

Each *request* (class `CRcRequest`) is associated with a **resource**, **a value** (no state) and a *request identifier (ID)*. The ***request ID*** allows to later modify or delete the request from any host.

Additionally, a request may have optional attributes as follows. In brackets, the argument name for `RcSetRequest` (Python API, Resources / General) and syntax for a definition string are shown.

Priority (*priority* / `*<prio>`)
: This is a number between 0 (= lowest, `rcPrioMin`) and 9 (= highest, `rcPrioMax`). The normal priority is 3 (`rcPrioNormal`). If multiple requests at the same priority exist, the oldest one dominates. (This behavior is required to allow the implementation of Mutexes, where lock owners cannot be preempted.)

***Start time* (or *on time*)** (*t0* / `[+<rep>]+<time>`)
: The request becomes active then, but is not considered before. If the optional repetition clause *[+<rep>]* is supplied, it defines a repetition interval (see *repeat* below).

***End time* (or *off time*)** (*t1* / `-<time>`)
: The request becomes inactive at this time and will be discarded then (unless the *repeat* attribute is set – see below).

Repetition interval (*repeat* / see *start time*)
: If set and the end time (*t1*) is reached, both *t0* and *t1* are incremented by this instead of removing the request. This allows a request to become active repeatedly for some time.

Hysteresis (*hysteresis* / `~<hyst>`)
: This allows the *Resources* library to postpone the requested action by up to *<hyst>* milliseconds in order to avoid switches forth and back again (for example, shutting down a computer that is needed again soon).

  If a positive hysteresis is given, a request will only be activated if no change to a different value is planned (according to existing requests) within the hysteresis time. Only follow-up events without a hysteresis or a hysteresis small enough that they cannot be pushed out of the current hysteresis interval are considered.

The complete syntax of string-based attributes is listed in Section 5.9.

If no requests are active for a resource, its value is left unchanged. It is legal to set the start and end times to identical values, in which case the value is set once at the specified time (given that no other request with a higher priority prevents that) and the request is deleted again.

Resources of type *trigger* are processed specially: Only at the on time, trigger events are generated. Hence, it is recommended to always supply an end time for triggers (which may be equal to the start time) to auto-remove the request.

Examples

1. The user pushes a button to open the window shades. One second later, an automatic script decides that the shades should be closed. What should happen now? An answer to this question can only be given by the end user. However, the *Home2L's* request mechanism allows to implement various different solutions.

   Possible solution A:

   - The automatic script sets permanent requests *#script* for a certain value (e. g. 0 for "up" and 1 for "down") and a priority of 3 (the default rules priority).
   - Pushing one of the user buttons "up" and "down" generates a timed requests *#user* with the off time attribute set to some time in the future (e. g. 1 hour after the button was pushed) and an increased priority of 7. This request will dominate over the script's request, and after the off time, the script request will take over again.

   Possible solution B:

   - Same as solution A, but with the requests generated with the "up" and "down" button being permanent.
   - A third push button is used with a "stop manual mode" functionality. Pushing it removes the *#user* request.

2. A PC-based video recorder can be requested to be on by recording timers as well as manually by the user to watch TV. Whenever unused, the PC should shut down automatically, but only if the time of the next recording is more than 30 minutes in the future to avoid unnecessary power cycles.

   This can be modelled as follows:

   - The video recorder sets timed requests *#timer* (with appropriate start and end times) for its recordings with a boolean value of '1'.
   - The user sets and deletes untimed manual requests *#user* with a value of '1'.
   - To let the computer shut down if not needed, a permanent request *#default* with a low priority and a value of '0' is set with a hysteresis of 30 minutes.

5.7. The *Shell* – The Command Line Interface to *Resources*

The *Home2L Shell* is a command line interface to the *Resources* library and serves as a "swiss army knife" to access and inspect the resources and servers of a *Home2L* cluster.

With the *Home2L Shell* you can:

- list all (server) hosts in the cluster and check their status and availability,
- list and inspect the resources directory,
- for each resource, see its current value and state, its current list of subscribers and all currently active requests,
- manually set and delete requests,
- monitor resource events,
- load and run drivers,
- get information on *Home2L* itself (e.g. the list of available value types).

The *Home2L Shell* can be run interactively or execute commands in batch mode. The latter can be used to set/delete requests from a shell script and for data logging.

Details on the usage of the *Home2L Shell* can be obtained by the 'help' command:

```
$ home2l shell
home2l> help
```

Details on the batch usage can be obtained by:

```
$ home2l shell -h
```

Section 2.4 contains detailed examples for working with the *Home2L Shell*.

5.8. Writing Drivers

Drivers can be can be packaged as loadable modules or be provided by an application.

Loadable drivers can be loaded and run by any *Home2L* instance linked against the *Resources* library. They are loaded if a respective `drv.<id>` configuration entry is set. A loadable driver can either be a binary shared object (see Section 5.8.1) or a script communicating with the *Resources* library by its standard input and output (see Section 5.8.2).

A good way to test a loadable driver is to use the `home2l-shell`, which may but does not necessarily have to run a server for this:

```
$ home2l shell drv.mydriver=path/name/of/mydriver
```

Application-hosted drivers are instantiated by an applications. A good example is the driver of `home2l-wallclock`, which this way exposes certain resources of the *WallClock* application such the Bluetooth state, the display mode or an option to mute the music player. Application-hosted drivers can be implemented in *Python* or in *C/C++*.

5.8.1. Binary Drivers (Loadable)

Examples: *Demo (drivers/demo/), GPIO (drivers/gpio/)*

A binary driver is written in native C/C++ code and compiled as a shared object (*.so*) file, which at run time can be loaded by any application as directed by the respective `drv.<id>` configuration entry.

The driver must contain a single entry function declared as follows:

```
HOME2L_DRIVER(<name>)
(ERcDriverOperation op, CRcDriver *drv, CResource *rc, CRcValueState *vs) {
  switch (op) {
    case rcdOpInit:
      ...
      break;

    case rcdOpStop:
      ...
      break;

    case rcdOpDriveValue:
      ...
      break;
  }
}
```

As outlined here, the driver must provide up to three operations:

The "Init" operation registers all resources and initializes the driver itself.

The "Stop" operation is called just before the driver gets unloaded. It must stop/join all own background threads and shut down its own operations. Resources (`CResource` objects) do *not* have to be unregistered, this will be done automatically later.

The "DriveValue" operation is only called and required for writable resources. It must drive a new value to the real device (e.g. some actor) to make something happen.

It is *neither necessary nor allowed* to call any `CResource` method here. The sole task of the driver is to operate its hardware. By default, the driven value with a state of "valid" is reported automatically. If this is not appropriate (e.g. due to an error or if the actor requires some time to fulfill the requested action), the passed `CRcValueState` object can and should be modified accordingly. Please consult the code documentation (Resources / Drivers) for details about that.

> ***i*** For example, a driver for window shades which is requested to close the shades would now modify the `CRcValueState` object and set its state to "busy" to indicate that the shades are now moving down. Later, when they are actually closed, the driver calls `CResource::ReportValue()` with a value/state of "1/valid" (assuming the value for closed shades is "1") to report the action was completed successfully.

The reporting of values for sensor-like hardware can be done asynchronously at any time from any thread by calling one of the `CResource::ReportValue()` methods (see the C/C++ API for details). A typical driver starts its own background thread during initialization which communicates with the hardware and reports value or state changes as adequate. After the `rcdOpStop` call, no reporting is allowed any more.

Further information on drivers can be found in the code documentation (Resources / Drivers).

5.8.2. Script Drivers (Loadable)

Example: *Weather (`drivers/weather/`)*

A *script-based* driver is typically implemented as a shell script and communicates with the *Resources* library by its standard input and output by means of simple text lines. To get an impression of the simple protocol, you may call such a driver on the command line, for example:

```
$ $(HOME2L_ROOT)/lib/home2l-drv-weather
```

At the script developer's choice, the driver can be operated in one of two modes:

a) *Keep-running mode:* The script is running permanently (and eventually restarted automatically if it crashes).

b) *Polling mode:* The script is run for individual polling or driving operations and is expected to terminate as soon as the respective operation is completed.

Script Invocation and Input to the Driver Script

The driver script is called by the *Resources* library with one of the following arguments:

```
<script> -init     # Initialize driver, driver must report its properties (see below).

<script> -restart  # Restart driver (only after abnormal stop in keep-running mode);
                   # The driver does not need to report anything.

<script> -poll     # Driver is polled for new readable values (only in polling mode)
                   # and must report updates by "v" messages (see below).

<script> -drive <resourceLID> <value>
                   # Drive a value (only in polling mode);
                   # The driver must report the success by means of a "v" message
                   # indicating the new value and state.
```

In "keep running" mode, values to drive are passed to the script's standard input as lines with the following format:

```
<resourceLID> <valueState>
```

The resource's *local ID (LID)* is part of the *URI* following the driver name.

Expected Script Output

During initialization (*'-init'* call), the driver is expected to output a series of lines with the following contents:

```
d <resourceLID> <type> (ro|wr) [ <value> [ <reqAttributes> ] ]
                   # Declare a resource. If the optional argument <value> is supplied,
                   # a default request is set using \refapic{CResource::SetDefault()}.

p <pollInterval>   # Define the polling interval in seconds
                   # (0 = no polling; Default = no polling)

.                  # Initialization complete - enter polling mode
:                  # initialization complete - enter "keep running" mode
```

The initialization phase must be completed as quickly as possible and end with the output of either `.` or `:`.

In the active phase, lines with the following contents can be output:

```
v <resourceLID> <valueState>  # Report a new value/state.
p <pollInterval>              # Change the polling interval (polling mode only).
```

Syntax Reference

`<resourceLID>` : Local ID of a resource.

The resource's *local ID (LID)* is the part of the *URI* following the driver name.

`<type>` : A type specification – see Section 5.3.2.

`<value>` : A value – see Section 5.9.

Note: Boolean values passed to the driver are always written as '0' or '1'.

`<valueState>` : A value with optional state attribute – see Section 5.9.

`<reqAttributes>` : Request attributes – see Sections 5.6 and 5.9.

`<pollInterval>` : Polling interval in seconds as a decimal integer number.

5.8.3. Drivers in C/C++ Appplications

Examples: `home2l-wallclock` (`wallclock/system.C`),
`home2l-brownie2l` (`brownies/home2l-brownie2l.C`)

Drivers in applications are set up by instantiating an object of class `CRcDriver` or a subclass thereof and registering the object by calling `CRcDriver::Register()` or `RcRegisterDriver()`. The registration must happen in the *elaboration phase*, which is between the calls to `RcInit()` and `RcStart()`.

Details are described in the code documentation (Resources / Drivers).

5.8.4. Drivers in Python Appplications

Example: *The ShowHouse (`resources/home2l-showhouse`)*

Drivers can be registered as part of a Python application.

> ***i*** The demo `home2l-showhouse` implements a driver for a) a set of (simulated) sensors and actors for a virtual building, but also b) an asynchronous keyboard driver for its own UI operation. The latter demonstrates how to work with background threads in Python.

A driver is defined using the decorator

```
@newDriver ( <driver name>, [ <success state> ] )
def DriverFunc (rc, vs):
  ...
```

and resources for it are registered by the function

```
RcNewResource (driverName, resourceLID, rcType, writable)
```

To improve code readability, it is allowed to place `RcNewResource()` invocations before the driver is registered, in which case the resources are registered later together with the driver. Both drivers and their resources *must* be registered during the elaboration phase before `Home2lStart()` or `Home2lRun()` is called.

The driver function *DriverFunc()* is responsible for driving values to writable resources and is the similar to the `rcdOpDriveValue` operation in a binary driver. Different from binary drivers, the Python driver function is allowed to call `CResource::ReportValue()` to indicate its current value.

Different from binary drivers, a Python driver is always called synchronously from the main Python thread (from `Home2lRun()` or `Home2lIterate()`) in a deferred way. Before this, immediately after the internal "Drive" operation, a state change towards "busy" is reported automatically. This automatic reporting behavior can be modified by the optional *successState* argument of *@newDriver* decorator (see `NewDriver()` for possible options).

Value and state changes can be reported any time by the `CResource::ReportValueState()` method. If only the value changed, `CResource::ReportValue()` can be used instead. With the Python API, the `CResource::Report...()` methods are the only ones that can be called from any thread. All other API functions must be called from the main thread.

5.9. Syntax of Value/State and Request Specifications

In some places, particularly in the `home2l-shell` and the Python API, values, states or requests are printed or accepted in textual form. This section gives a reference on the syntax for value/state objects and requests.

The syntax of a **value/state** object is:

```
<valueStateFull> ::= [ "(" <type> ")" ] <valueState> [ @<time> ]

<valueState> ::= [!]<value>|?

  <value>    : State is 'valid'.
  !<value>   : State is 'busy'.
  ?          : State is 'unknown'.

<value> ::= <bool> | <int> | <float> | <string> | <time> | <unitval> | <enumval>

  <bool>    ::= [0fF] | [1tT+]              : Boolean value
  <int>     ::= [-][0-9]+                   : Integer value
  <float>   ::= [-][0-9]*[.[0-9]+][E[+/-][0-9]+]  : Floating-point value
  <string>  ::= [0-9a-zA-Z\]+ | \0          : String (\-escaped, UTF8 encoding)
  <time>                                    : Time value (see below)
  <unitval> ::= [<float>|<int>]<unit>       : Unit value (<unit> is the unit string)
  <enumval> ::= [_a-zA-Z][_a-zA-Z0-9]+      : Enumeration value
```

A *<valueState>* expression must not contain any spaces. Type and timestamp attributes in *<valueStateFull>* are separated by spaces.

Requests are specified as follows:

```
<request> ::= <value> [ <reqAttributes> ]

<reqAttributes> is a space-separated subset of:

  #<id>           : Request ID [default: instance name]

  *<prio>         : Priority (0..9) [default: 3]

  [+<rep>]+<time> : Start time and optionally repeat interval.
                    <rep> may be empty (= repeat daily), or a <time> value as described
                    below. e.g. "2d" for 2 days. A common case is to have a daily
                    repetition, in which case just an additional '+' needs to be added
                    to the start time. For example, to turn on some resource daily
                    from 5 to 7 p.m., enter "++17:00 -19:00"

  -<time>         : End time.

  ~<hyst>         : Hysteresis in milliseconds
```

Time values *<time>* – either as a value or a an start/end/repeat time specification – can be specified in several alternative formats:

```
YYYY-MM-DD[-hhmm[ss[.<millis>]]] : Date and time, interpreted as local time

t<unsigned integer>   : Absolute time in milliseconds since the Epoch (POSIX time)

<integer>[<unit>]     : Relative time in milliseconds or some other unit with, if <unit>
                        is supplied. Possible units are: seconds ('s'), minutes ('m'),
                        hours ('h'), days ('d'), and weeks ('w').

hh:mm[:ss[.<millis>]] : Time relative to 0:00 today; hh may be > 23 to specify a time
                        in the coming day(s).
```

With the evolution of the *Home2L* suite, the information above may become outdated or incomplete. The most up-to-date information can be found in the following places of the C/C++ API documentation:

- `CRcValueState::SetFromStr()` (source file: `resources/resources.H`)
- `CRcRequest::SetFromStr()` (source file: `resources/resources.H`)
- `TicksFromString()` (source file: `common/base.H`)

5.10. List of Configuration Parameters

5.10.1. Parameters of Domain `rc`

`rc.config` (string) [= "resources.conf"] `resources/rc_core.C:76`

Name of the Resources configuration file.

`rc.enableServer` (bool) [= false] `resources/rc_core.C:80`

Enable the Resources server.

(Only) if true, the Resources server is started, and the local resources are exported over the network.

`rc.serveInterface` (string) [= "any"] `resources/rc_core.C:88`

Select interface(s) for the server to listen on.

If set to "any", connections from any network interface are accepted.

If set to "local", only connection attempts via the local interface (127.0.0.1) are accepted. This may be useful for untrusted physical networks, where actual connections are implemented e.g. by SSH tunnels.

If a 4-byte IP4 address is given, only connections from the interface associated with this IP address are accepted. This way, a certain interface can be selected.

This value is passed to bind(2), see ip(7) for more details. The value of "any" corresponds to INADDR_ANY, the value of "local" corresponds to INADDR_LOOPBACK.

`rc.network` (string) [= "127.0.0.1/32"] `resources/rc_core.C:105`

Network prefix and mask for the Resources server (CIDR notation).

Only connections from hosts of this subnet or from 127.0.0.1 (localhost) are accepted by the server.

`rc.maxAge` (int) [= 60000] `resources/rc_core.C:112`

Maximum age (ms) tolerated for resource values and states.

If a client does not receive any sign of life from a server for this amount of time, the resource is set to state "unknown" locally. Servers send out regular "hello" messages every 2/3 of this time. Reducing the value can guarantee to detect network failures earlier but will increase the traffic overhead for the "hello" messages.

This value must be consistent for the complete Home2L cluster.

`rc.netTimeout` (int) [= 3000] `resources/rc_core.C:124`

Network operation timeout (ms).

Waiting time until a primitive network operation (e.g. connection establishment, response to a request) is assumed to have failed if no reply has been received.

`rc.netRetryDelay` (int) [= 60000] `resources/rc_core.C:131`

Time (ms) after which a failed network operation is repeated.

Only in the first period of `rc.netRetryDelay` milliseconds, the connection retries are performed at faster intervals of `rc.netTimeout` ms.

`rc.netIdleTimeout` (int) [= 5000] `resources/rc_core.C:137`

Time (ms) after which an unused connection is disconnected.

`rc.relTimeThreshold` (int) [= 60000] `resources/rc_core.C:141`

Threshold (in ms from now) below which remote requests are sent with relative times.

This option allows to compensate negative clock skewing effects between different hosts. If timed requests are sent to remote hosts, and the on/off times are in the future and in less then this number of milliseconds from now, the times are transmitted relative to the current time. This way, the duration of requests is retained, even if the clocks of the local and the remote host diverge. (Example: A door opener request is timed for 1 second and should last exactly this time.)

`rc.userReqId` (string) [= "user"] `resources/resources.C:33`

Request ID for user interactions, e.g. with the WallClock floorplan or with physical gadgets.

`rc.userReqAttrs` (string) [= "-31:00"] `resources/resources.C:37`

Request attributes for for user interactions.

This parameter defines the attributes of requests generated on user interactions, e.g. with the WallClock floorplan or with physical gadgets.

The probably most useful attribute is the off-time. For example, if the attribute string is "-31:00" and a user pushes a button to close the shades, this overrides automatic rules until 7 a.m. on the next morning. Afterwards, automatic rules may open them again.

The request ID must be defined by setting `rc.userReqId` . Adding an ID field to the attributes here has no effect.

`rc.maxOrphaned` (int) [= 1024] `resources/resources.C:841`

Maximum number of allowed unregistered resources.

Resource objects (`class CResource`) are allocated on demand and are usually never removed from memory, so that pointers to them can be used as unique IDs during the lifetime of a programm. Unregistered resources are those that presently cannot be linked to real local or remote resource. They occur naturally, for example, if the network connection to a remote host is not yet available. However, if the number of unregistered resources exceeds a certain high number, there is probably a bug in the application which may as a negative side-effect cause high CPU and network loads.

This setting limits the number of unregistered resources. If the number is exceeded, the application is terminated.

`rc.timer` (bool) [= true] `resources/rc_drivers.C:55`

Enable/disable the 'timer' driver.

`rc.drvMinRunTime` (int) [= 3000] `resources/rc_drivers.C:350`

Minimum run time of a properly configured external driver (ms).

To avoid endless busy loops caused by drivers crashing repeatedly on their startup (e.g. due to misconfiguration), a driver crashed on startup is not restarted immediately again, but only after some delay.

This is the time after which a crash is not handled as a startup crash.

`rc.drvCrashWait` (int) [= 60000] `resources/rc_drivers.C:359`

Waiting time (ms) after a startup crash before restarting an external driver.

To avoid endless busy loops caused by drivers crashing repeatedly on their startup (e.g. due to misconfiguration), a driver crashed on startup is not restarted immediately again, but only after some delay.

This parameter specifies the waitung time.

`rc.drvMaxReportTime` (int) [= 5000] `resources/rc_drivers.C:368`

Maximum time (ms) to wait until all external drivers have reported their resources.

`rc.drvIterateWait` (int) [= 1000] `resources/rc_drivers.C:371`

Iteration interval (ms) for the manager of external drivers.

5.10.2. Parameters of Domain `drv`

`drv.<id>` (string) `resources/rc_drivers.C:327`

Declare/load an external (binary or script-based) driver.

The argument <arg> may be one out of:

a) The name of a driver .so file (binary driver).

a) The invocation of a script, including arguments.

a) A '1', in which case <id> is used as <arg> (shortcut to enable binary drivers).

a) If set to '0', the driver setting is ignored.

Relative paths <name> are searched in:

- <HOME2L_ROOT>/etc[/<ARCH>]
- <HOME2L_ROOT>/lib/<ARCH>/home2l-drv-<name>.so
- <HOME2L_ROOT>/lib/<ARCH>/home2l-drv-<name>
- <HOME2L_ROOT>/lib[/<ARCH>]
- <HOME2L_ROOT>/

Please refer to the section on writing external drivers in for further information on script-based drivers.

5.10.3. Parameters of Domain `shell`

`shell.historyFile` (string) [= ".home2l_history"] `resources/home2l-shell.C:36`

Name of the history file for the home2l shell, relative to the user's home directory.

`shell.historyLines` (int) [= 64] `resources/home2l-shell.C:39`

Maximum number of lines to be stored in the history file.

If set to 0, no history file is written or read.

`shell.stringChars` (int) [= 64] `resources/home2l-shell.C:44`

Maximum number of characters to print for a string..

If set to 0, strings are never abbreviated.

5.10.4. Parameters of Domain `location`

`location.latitudeN` (float) [= 48.371667] `resources/rc_drivers.C:64`

WGS84 coordinate (latitude north) of the building.

This value is (amoung others) used by the 'timer' driver for twilight calculations.

`location.longitudeE` (float) [= 10.898333] `resources/rc_drivers.C:70`

WGS84 coordinate (longitude east) of the building.

This value is (amoung others) used by the 'timer' driver for twilight calculations.

6. Writing Automation Rules

6.1. Overview

With the *Home2L* suite, automation scripts are typically written in Python. Automation scripts are normal Python programs. They can run on any machine, there can be multiple of them, and they can be started or stopped any time. The latter is particularly useful for testing and debugging. Even interactive work in a Python shell is possible.

For a quick start, it is recommended to read the commented sample rules file `rules-showhouse` used in the tutorial (Chapter 2). Also, it is worth looking into the source code of `home2l-showhouse`, which is implemented like a rules script in Python (see Sections 2.5 and 2.9).

Typically, a rules script performs some initialization, declares some triggered functions, and finally calls `Home2lRun()`) to enter the main event loop of the *Home2L* library/package. While the native *Home2L* library makes use of multi-threading, all Python functions besides `CResouce.Report...()` (see Section 5.8.4) must be called from the main thread.

The following subsections give some additional hints and references to the respective places in the Python API documentation, which serves as a reference manual for rules writing.

6.2. Triggered Functions

6.2.1. Resource Events

Functions to be run on each event related to a specific set of resources can be defined with the `@onEvent()` decorator:

```
@onEvent ( <resources> )
def MyFunc (ev, rc, vs):
  # user code
```

Functions decorated this way are called for each individual event. This allows to receive all events without exceptions in the correct order (important to not miss any event or quick value change). Events can be value/state changes as well as indications whether the connection to a (remote) resource has been established or lost.

As arguments, the decorator expects a set of resources for which events should be delivered. The argument may be a single resource, a tuple, or a list. Each resource may be specified by a string with its URI or by a reference to the resource object previously retrieved by `RcGet()`.

The argument *'ev'* passed to the user function identifies the type of event, which can be:

- `rceValueStateChanged`: The value or state has changed.
- `rceConnected`: The connection has been (re-)established.
- `rceDisconnected`: The connection has been lost.

The arguments *rc* and *vs* are the resource and their current value and state (see class `CRcValueState`), respectively.

> *i* To maintain the precise event model, it is important to always use the passed argument *vs* and to never query *rc* for its value/state, which may have changed since the event occurred.

6.2.2. Value/State Updates

Functions decorated with `@onUpdate()` are called whenever the value or state of a resource changes:

```
@onUpdate ( <resources> )
def MyFunc ():
  # user code
```

If the value or state changes very quickly, the user function may be called only for the latest, most up-to-date value/state. Unlike functions decorated with `@onEvent()`, events may automatically be dropped to optimize performance. This decorator is recommended if it is sufficient to always have the most up-to-date value/state of a resource, whereas the precise sequence of events is not relevant.

> *i* *MyFunc()* has no arguments here. Unlike the precise event model, with the "always up to date" model assumed here, it is not critical to read the value/state of some resource inside the user function. The worst thing that can happen is that a value newer than the value that caused the invocation of the user function is retrieved.

6.2.3. Timed Functions

To run a function at a certain time or at certain intervals, the `@at()` decorator can be used:

```
@at ( t = <t> [ , dt = <interval> ] )
def MyFunc ():
  # user code
```

The *dt* argument is optional. If unset (*None* or 0), the function is called once only.

Sometimes, a timed function is to be run multiple times in different situations. The reuse of the user function (*MyFunc()*) is simplified, if

```
RunAt (func, t, dt = 0, data = None)
```

is used instead of the decorator.

6.2.4. Daily Requests

Often, static requests need to be set that are re-calculated on a daily basis. For example, an outdoor light might need to be switched on at night depending on the current sunset and sunrise times. For such cases, *daily rules* may be defined, which set some requests accordingly.

The following decorator defines a function for such daily requests.

```
@daily ( <host set> )
def MyFunc (host):
  ...
```

As an argument, the decorator expects a set of host IDs, which is a list or tuple of strings. The host set is used to monitor the availability of the hosts and eventually also run the user function after a host has crashed or being restarted and becomes reachable (again).

The user function *MyFunc()* is called with the host ID as an argument on each day, shortly after midnight. It is called once per host, so that it should only set requests for the host identified by the argument.

Note: For getting the time and to detect the presence or absence of a host, the `@daily` decorator subscribes to the `timer/daily` resources of each host of the host set. If no host is passed, `/local/timer/daily` is subscribed. Hence, it is required that the *timer* driver is not disabled on any of the hosts.

6.3. Retrieving Resource Values and States

6.3.1. Value-Only Retrieval

The value of a `CResource` object *"rc"* can be retrieved by

```
val = rc.Value()                  # returns 'None' if state is 'rcsUnknown'
```

If the state is *rcsUnknown*, *None* is returned. A destinction between the states *rcsValid* and *rcsBusy* is not possible.

The caller must be prepared that calls of the `Value()` method may return *None* any time. Alternatively, to simplify rules development, the method

```
val = rc.ValidValue ( <default> ) # returns <default> if state is 'rcsUnknown'
```

can be used instead. If the actual value is unknown, the passed default is returned, and the caller does not have to check for the *None* return value explicitly.

6.3.2. Retrieving the Value, State and Attributes

The value, state and additional attributes (time stamp, explicit type) can be retrieved by:

```
vs = rc.ValueState()
```

The returned CRcValueState object *"vs"* contains a copy of all information on the type, the value, and the state of the resource.

The value itself can then be retrieved by:

```
val = vs.Value ()                 # returns 'None' if state is 'rcsUnknown'
val = vs.ValidValue ( <default> ) # returns <default> if state is 'rcsUnknown'
```

The following methods deliver additional attributes of CRcValueState objects:

```
vs.Type ()      # the type
vs.State ()     # the state
vs.TimeStamp () # the time stamp
vs.IsValid ()   # True, if the state is 'rcsValid', neither 'rcsBusy' nor 'rcsUnknown'.
vs.IsKnown ()   # True, if the value can be retrieved (state is 'rcsValid' or 'rcsBusy')
```

Due to the highly concurrent nature of the *Resources* library, resource values and states may change any time. In particular, multiple successive calls of CResource.ValueState() or CResource.Value() may deliver different results. The method CResource.ValueState() delivers a consistent local copy of the complete value/state object, so that it is safe to read multiple attributes from *"vs"*afterwards.

To functions decorated with @onEvent(), an appropriately captured value/state object is already passed by the *"vs"* argument, and that must be used in order to avoid race conditions. Event-triggered functions accessing their resources to read values most probably have a design flaw and are prone to race conditions!

6.4. Placing requests

Requests can be set or deleted by:

```
RcSetRequest (rc, <id, value and attributes> )
RcDelRequest (rc, id)
```

The *rc* argument can either be a string with the URI or a reference to the resource object previously retrieved by RcGet(). The remaining arguments of RcSetRequest() allow to pass the requested value and any desired request attributes in a very flexible way either as direct values or as definition strings as accepted by the home2l-shell. Also, a mixture of direct values and a definition string is allowed.

For example, an integer value of 7 with default attributes can be requested in any of the following ways:

```
RcSetRequest (rc, 7);
RcSetRequest (rc, "7");           # request definition string only containing the value
RcSetRequest (rc, reqDef = "7");  # same with explicit argument
RcSetRequest (rc, value = "7");
```

An integer value of 7 with with priority 3 can be requested in any of the following ways:

```
RcSetRequest (rc, 7, priority = 3);  # direct arguments
RcSetRequest (rc, 7, "*3");          # direct value, attribute by string
RcSetRequest (rc, "7 *3");           # both together by a request definition string
```

Strings should always be explicitly assigned to the *value* argument in order to avoid them being misinterpreted as definition strings. For example, a string value of "Hello *3" with priority 5 and an off time at 6pm today can be requested as follows:

```
RcSetRequest (rc, value = "Hello *3", attrs = "*5 -18:00");
RcSetRequest (rc, value = "Hello *3", prio = 5, t1 = "-18:00");
```

Please consult the Python API documentation of `RcSetRequest()` and `RcDelRequest()` for details and more examples. The request attributes are explained in Section 5.6.

7. *Brownies* – Helpful Microcontrollers for Sensors and Actors

7.1. Preamble: What is a *Brownie*?

"A brownie [...] is a household spirit from British folklore that is said to come out at night while the owners of the house are asleep and perform various chores and farming tasks." [Wikipedia: Brownie (folklore)]

Brownies in mythology are said to be

- helpful,
- mostly invisible,
- they do not want a salary except for a bowl of milk,
- they leave the home if insulted.

Home2L Brownies are

- helpful,
- mostly invisible,
- based on AVR 8-bit microcontrollers – they do not want much electrical power except for a few milliwatts,
- they get insulted if somebody claims that a 16-bit or 32-bit microcontroller can do a better job than they do.

Unlike brownies in mythology, *Home2L Brownies* come in groups: Installing multiple microcontrollers per room and tens or even a hundred of them in a house remains cheap in terms of both hardware costs and power consumption.

And they do not leave the home if insulted: In the (rare) case they crash and do not respond, they can be power-cycled, configured and re-flashed with software updates remotely. The possibility of remote updating is generally useful if they are installed in places difficult to reach such as switches or installation boxes.

Home2L Brownies are small microcontroller boards suitable to connect standard sensors like temperature sensors, window sensors, switches and buttons as well as some actuators to a Linux host – similar to a KNX installation.

7.2. Overview

The *Home2L Brownies* project provides a protocol, a microcontroller firmware, and a maintenance tool for simple do-it-yourself sensor and actor devices.

The devices (*"Brownies"*) are based on *AVR ATtiny* microcontrollers. Presently supported models are *ATtiny84*, *ATtiny85*, and *ATtiny861* and their variants (e.g. *ATtiny84a*). These differ in the number of pins, but else have very similar features: 8 KB of program memory, 512 bytes of EEPROM, 512 bytes of RAM, a universal serial interface (USI), ADC(s), timers.

Brownie devices are interconnected and connected to a Linux host using a tree of two-wire interface (TWI) buses (see Section 7.3). The *Home2L* bus is based on *i2c*, which allows very simple circuitry. Neither an external crystal oscillator nor a dedicated transceiver is needed. Just two resistors, the microcontroller, and eventually a blocking capacitor are sufficient to build a *Brownie* device.

For the interconnections, a 4-wire cable (e.g. KNX) is suitable for the power supply and TWI communication. Such cables become more and more common in new buildings. Compared to wireless connections, cable connections are generally more reliable and less prone to security attacks.

The *Brownie* firmware is modular. Besides some base functionality and the TWI slave communication stack, each firmware instance implements a customized set of device features such as support for GPIOs, temperature sensors, or shades controllers (see Sections 7.4 and 7.7).

On the Linux host side, a powerful adminstration tool (`home2l-brownie2l`) allows to orchestrate, configure and maintain the *Brownie* bus tree (see Section 7.5). Sensors and actors are published to the *Home2L* cluster as resources by the `home2l-drv-brownies` driver (see Section 10.4).

In summary, the general goals of the *Brownies* subproject are:

- simple hardware,
- eventual shortcomings of simple hardware compensated by sophisticated software,
- low power consumption, low hardware costs,
- reliability and maintainability.

Why 8-bit microcontrollers?

Using 8-bit *ATtiny* microcontrollers has a number of advantages:

- The hardware is simple, robust and mature.
- The computational power is absolutely sufficient for tasks like hosting switches, actors or temperature sensors.
- They are available in DIP packages and can be soldered manually – ideal for do-it-yourself projects.
- They are power-efficient.

- Security: 8 KB of program memory are perfectly sufficient for the communication stack, the device features, and a second maintanance system. However, it does not leave much space for malware.
- They have an integrated oscillator and a wide power supply range. This allows to build very simple circuits without any crystal oscillator or power converter running at 5V. Two resistors (and perhaps blocking capacitor) together with the microcontroller are sufficient to build a *Brownie* device.

People may ask for an evaluation board. There is none. It is not needed. A breadboard, an AVR programmer and a Linux machine with an *i2c* interface are sufficient to start own projects – see Section 2.6!

Why is the *Home2L* bus based on *i2c*?

Compared to asynchronous serial protocols like RS485, using an *i2c*-based protocol has some advantages:

- It is asynchronous, no precise clock (external crystal) is needed.
- No dedicated transceiver device is needed leading to simple circuitry.
- A host notification extension can be implemented without the necessity to support multi-mastering.

A limitation of *i2c* is that it has been designed for short distances, allowing only small wire capacitances. Additionally, if longer cables are used, the signal integrity may be affected by crosstalk effects and electrical disturbances.

These issues are compensated by operating the *Brownie* bus with smaller resistances and a lower bit rate than specified in the *i2c* standard (see Section 7.3.2). With these adaptations, the author could successfully connect two *Brownies* with a 100m KNX cable, which communicated at approx. 30 kBit/s with a negligible bit error rate.

A number of additional techniques help to cover longer (building-range) distances and to avoid transmission problems:

- *Brownies* can be cascaded using *hub* nodes. Critically long cables can be divided into an arbitrary number of smaller sections by inserting hubs.
- Bit rates of sub-trees can be adjusted and reduced to minimize errors.
- The protocol includes powerful CRC-based checksums for error detection and retransmission strategies.
- The protocol uses very short messages, leaving much room for the above measures such as bit rate reduction or retransmissions on errors. Small message sizes are also beneficial for a low power consumption.
- The resource driver (`home2l-drv-brownies`) collects detailed statistics about errors and theirs types to help identify problems in the cable installations and adjusting parameters.

Cost and Power Efficiency

Table 7.1 summarizes some characteristics and estimations related to power consumption, hardware costs and maintenance effort.

	Home2L TWI Bus	KNX
Bit rate [kBit/s]	approx. 30	9.6
Maintance software	Open Source (GPLv3)	Proprietary (ETS)
Standby power consumption / Device [1]	10 mW	150 mW – 240 mW
Standby power consumption / 50 Devices	0.5 W	7.5 W – 12 W
Hardware Costs / Device [2]	€2 – €10	€50 – €300
Hardware Costs / 50 Devices	€100 – €500	€2,500 – €15,000

Table 7.1.: Estimation of power consumption and hardware costs

[1] *Home2L TWI: 2mA at 5V (own measurement); KNX: 5-8 mA at 30V according to https://de.wikipedia.org/w/index.php?title=KNX-Standard&oldid=195615906*

[2] Very coarse estimation based on market prices as of 1/2020; Costs vary depending on the device type and features.

7.3. The *Home2L* Bus

7.3.1. Topology

Figure 7.1 shows an example topology for a *Brownie* network, referred to as a *bus tree*. The tree is controlled by a Linux host with an *i2c* interface, which is directly connected to a set of *root (or A-type) hubs*. The root hubs each have a *Home2L bus* master interface, to which multiple *Brownies* can be attached. Some of these may act as *intermediate (or B-type) hubs*, connecting other *Home2L* buses or *Brownie subnets*.

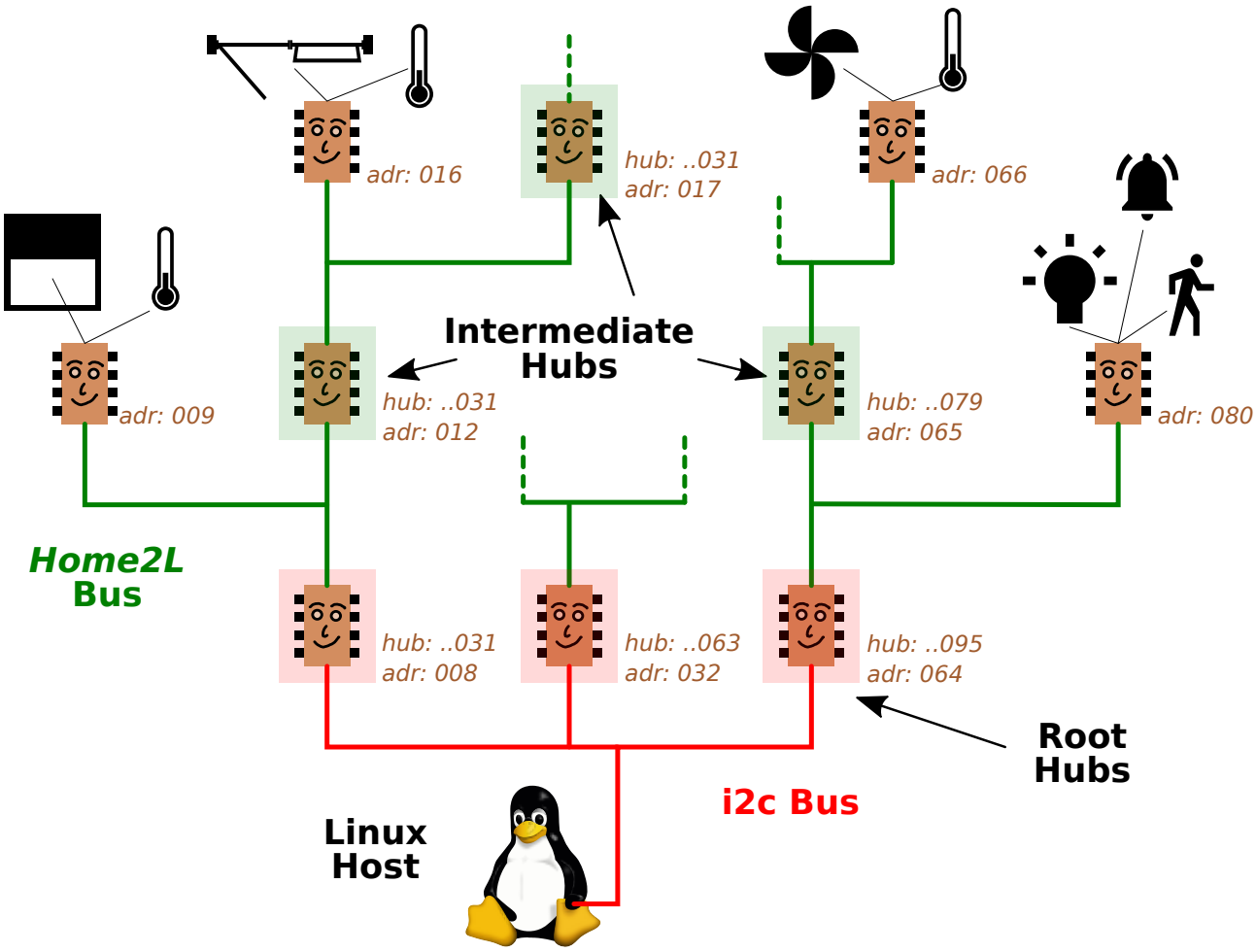


Figure 7.1.: Example topology of a *Brownie* network (tree)

Since *i2c* only allows small cable capacities, the root hubs should be installed close to the Linux host, ideally inside the same housing. The Linux host can be any computer with an *i2c* master interface supporting *i2c* clock stretching. Some versions of the popular *Raspberry PI* computers appear to have problems with clock stretching. Host devices known to work are the interfaces of *Allwinner A20* SoCs (*CubieTruck*, *Olimex A20-OLinuXino* series) or the *ELV USB-I2C* interface.

The *Home2L* bus cables can generally cover longer distances. They should have 4 wires for the TWI clock and data lines as well as GND and +5V for power supply. KNX cables are fine. Intermediate

hubs can be used a) to divide very long cables into smaller segments or b) to establish a tree-like topology eventually optimizing the efficiency of host notification (see Section 7.3.7). From a software perspective, hubs are transparent.

Technically, the *Home2L* bus is a two-wire interface (TWI), based on *i2c*, but with a few deviations allowing longer distances and host notification (see Section 7.3.2). In this book, the term TWI refers to any of a standard *i2c* or a *Home2L* bus. The difference between root hubs and intermediate hubs is that intermediate hubs are expected to be connected to *Home2L Brownies* only, while root hubs are fully *i2c*-compliant at their slave side.

Each *Brownie* is identified by a unique TWI address ("adr" in Figure 7.1). Hubs are configured with a maximum address (configuration option `hub_maxadr`, denoted "hub" in Figure 7.1), and they forward all messages addressed to some address larger than their own address and less or equal to `hub_maxadr` to/from their subnet over their TWI master port (upper side in the figure).

7.3.2. Electrical Characteristics

The *Home2L* bus is based on *i2c*, but deviates from the *i2c* specification in two points:

a) The recommended resistors and supply voltage result in (slightly) higher currents, and typical wire capacities exceed the limit set by the *i2c* standard.

b) When the bus is idle, slaves may pull the SDA line low in order to notify the master about an event such as a button press or a sensor value change (see Section 7.3.7).

The notification feature can be disabled (see Section 7.4), in which case the TWI slave interface becomes fully *i2c*-compatible (as for root hubs, see above).

Figure 7.2 shows the recommended circuitry. The supply voltage is 5V. Serial resistors of 100 Ω limit the switching peak current to 50 mA, which is the limit for the *ATtiny* (and many other) microcontrollers. The pullup resistors have 10 times the value of the serial resistors (1 kΩ), resulting in a drain current of 5 mA, which is slightly more than allowed for *i2c* busses (3 mA).

As for the wire assignments in cables, the SDA and SCL signals should be assigned to wires with maximum distance in order to minimize cross-talk effects.

Besides the host notification feature, the bus is a pure single-master bus. Clock stretching is used and must be supported by all devices. The bit rate is typically lower than the *i2c* rate of 100 kBit/s (which is no violation of the *i2c* specification). The default speed is approximately 30 kBit/s on *ATtiny* controllers running at 1 MHz. The rate can be reduced explicitly by the `hub_speed` configuration option (see Table 7.5).

7.3.3. Protocol

The *Home2L* bus protocol is a request-reply protocol. It is generic and not limited to *Brownies* and may also be used either outside *Home2L* or over other communication media.

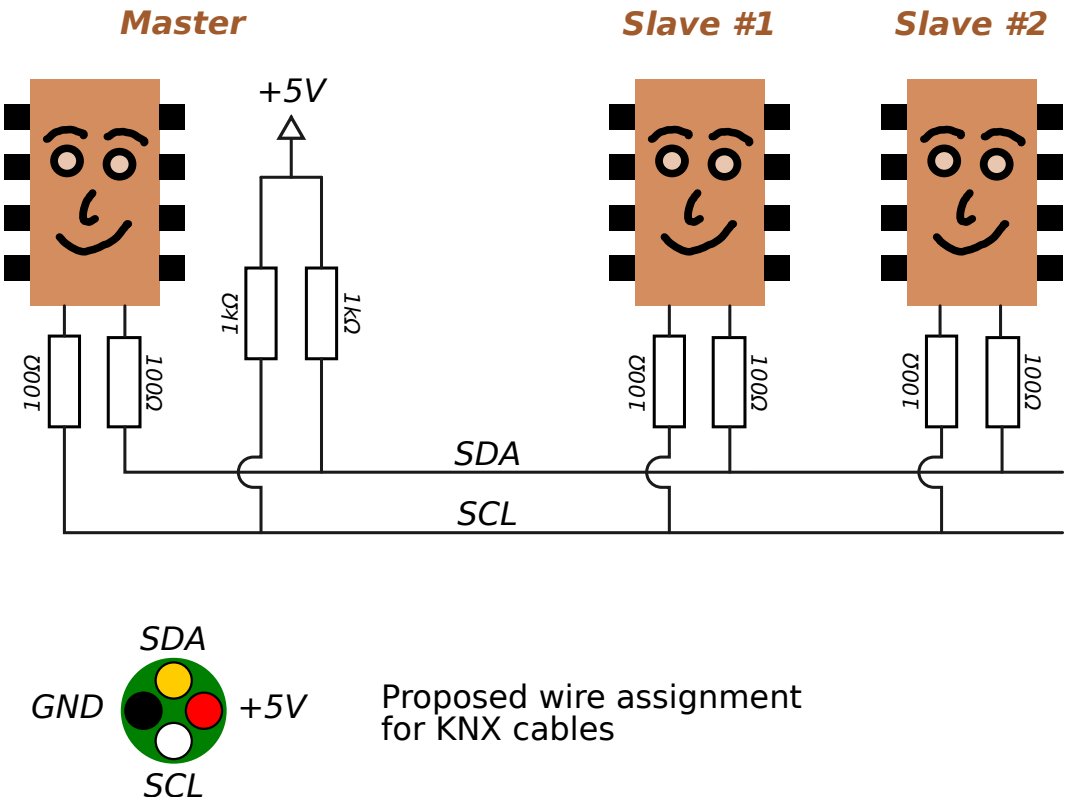


Figure 7.2.: Electrical characteristics of a *Brownie* TWI bus

This section describes the protocol in a generic way. Aspects specific to the implementation of *Home2L Brownies* are addressed in Section 7.4.

Each communication is initiated by the bus master and consists of a request message (a TWI write sequence) followed by a reply (a TWI read sequence). At present, 4 operations are defined, Figure 7.3 shows the request and reply sequences for each of them. The *register read/write* operations read or write one out of 64 virtual 8-bit registers, respectively. Typically, input/output resources are mapped to registers. The *memory read/write* allow to transfer larger chunks of data to or from a virtual memory. Data is aligned and transferred in units of 16 bytes. The address space allows to address up to 64 KB of (virtual) memory.

The register and memory maps are device-dependent and described in Section 7.4 for *Home2L Brownies*.

The communication is secured by strong CRC checksums, so that this protocol is suitable for noisy channels ("check8" and "check4" in Figure 7.3).

The first byte of each reply message contains a status code. Possible values are defined by the `EBrStatus` enumeration type.

The protocol permits *split transactions*, i.e. sending multiple requests to different devices before fetching all corresponding replies in order to accelerate multiple accesses and optimizing bus utilization.

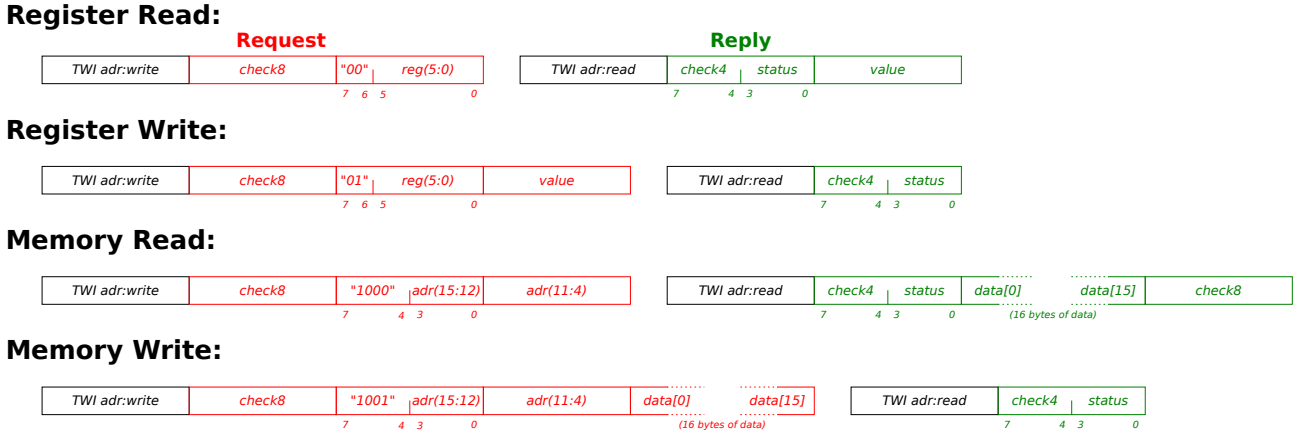


Figure 7.3.: *Brownie* protocol messages

7.3.4. Bus Addresses

Valid *Brownie* addresses range from 007 to 119, where 007 is reserved for freshly initialized, but unconfigured devices. By convention, in the *Home2L* context, TWI addresses are written as three-digit decimal numbers.

Table 7.2 lists the TWI addresses and their use. The rightmost column informally shows the use according to the *i2c* specification [https://www.i2c-bus.org/addressing]. This may be relevant if *i2c* devices should be connected to a *Home2L* TWI bus. High-speed devices and masters using the general calls (broadcasts) are generally incompatible with the *Home2L* TWI bus.

Address	*Home2L* TWI Bus	*i2c* Standard
000	(collision)	general call
001	(reserved)	CBUS addresses, no longer used by *i2c* standard
002	(reserved)	reserved for different bus formats
003	(reserved)	reserved for future purposes
004 – 006	(reserved)	high-speed master code
007	new *Brownie* (unconfigured)	
008 – 119	productive *Brownies*	
120 – 123	(reserved)	10-bit slave addressing)
124 – 127	(reserved)	reserved for future purposes)

Table 7.2.: TWI addresses used for the *Home2L* TWI bus

7.3.5. Hubs

Hubs receiving requests or reply fetches to a specified set of addresses (their *subnet*) must forward the messages to/from their slave TWI interface without modification. In case of an error, they should continue forwarding. They must not repeat any transmission, but they should change the status field of a reply from `brOk` to some error code if adequate.

Hubs can be cascaded arbitrarily. The only limitation for the number of subsequent hubs is the number of available addresses, since each hub requires an address for itself. In the current *Home2L* implementation, hubs forward their subnet requests and replies "on-the-fly", starting shortly after the addressing phase. Hence, each hub of a cascade adds a latency of just 2 bytes (plus a little processing time).

7.3.6. Error Handling

All messages are secured with 8-bit and/or 4-bit CRC checksums, so that the *Home2L* bus protocol is suitable for noisy channels. The checksum calculation algorithms, seeds and polynomials have been selected very carefully and, amoung others, fullfil the "HD-2" property, which means that up to two incorrect bits can be detected under any circumstances. Details can be found in the comments in `brownies/avr/interface.c`. For all possible requests and for almost all possible reply messages, the 8 or 4 bit checksums in the first bytes of the messages are sufficient for this. The "memory read" reply contains an additional 8-bit check field just for the data array, since for this message length a 4-bit checksum alone would be insufficient to fulfill the HD-2 property.

In case of an error, the root master may perform error correction by repeating the request and re-fetching the reply. Besides the root master (Linux host in Figure 7.1), no other component is allowed to repeat a message. In some cases, requests cannot be repeated for semantic reasons. For example, reading the register `BR_REG_CHANGED` auto-modifies the register (resets all bits). For such cases, methods like `CBrownie::Communicate()` have a 'noResend' option to disable automatic message repetitions. Errors can then be handled properly at the application layer.

Dealing with lost messages: Slaves generally never repeat a reply, and hubs generally never repeat a request towards or reply from their subnet. In the case that a slave is asked for a reply without having received a request before (e.g. due to a lost or incorrect request message), the slave replies with the status code `brRequestCheckError`. Requests with an invalid checksum are ignored by the slave. This way, it can be guaranteed that a reply received by the root master always belongs to the last request sent.

7.3.7. Host Notification

The *Home2L* bus is a single-master bus, and generally, the root master must poll all devices in regular intervals to check for changes. In some cases, e.g. for devices with push buttons or switch sensors, it is desirable that a slave can notify the master in order to improve responsiveness and to allow longer polling intervals. The *host notification* feature allows slave devices to draw a bus master's attention.

If the bus is idle (slaves must monitor start and stop conditions), a slave may pull the SDA line low for a certain time (e. g. 10 ms, at least the typical transmission time of two bytes). The master can detect this and can immediately start polling its slaves.

In order to identify the device which initiated the notification, the master must poll all its slaves. When using a tree-like topology with hubs, this process can be accelerated considerably: The hubs must implement a register with a flag indicating whether any device in its own subnet has submitted a notification. The root master can then poll the (primary) hubs first and only poll devices from subnets if this flag is set. In the case of *Home2L Brownies*, this mechanism is implemented with the `CHANGED_CHILD` bit of the `BR_CHANGED` register.

Collision handling: A *Home2L* bus master must check the SDA line before starting a transfer to avoid collisions. Slaves, and on the other hand, must wait for the bus to become idle before issuing a notification, usually by monitoring start and stop conditions. However, collisions can generally not be avoided completely. In case of a collision between a slave's notification and a master's addressing, the bus appears as if the master is addressing 000. Such messages must be ignored. For this reason, broadcasts are not allowed in a *Brownie* tree. Collisions between multiple slaves are not critical. Since SDA is an open-drain line, two notifications issued simultaneously appear as a single one on the bus. Hence, the master must poll all directly connected devices in case of a received notification.

In order to avoid bus contention, slaves should not repeat a notification (e.g. if it does not get polled within a certain time). Since collisions and the loss of notifications cannot be avoided completely, masters must poll all devices regularly anyway, even if no notification was detected.

Slaves performing host notification cannot be used on a standard *i2c* bus. For *Home2L Brownies*, the feature flag *"notify"* (see Table 7.3) indicates whether a device may issue notifications and is therefore not *i2c*-compliant. If this flag is not set, the device is *i2c*-compliant.

7.4. The Brownie Firmware

7.4.1. The Family

The *Brownie* firmware is modular and highly configurable, containing some base functionality and a set of selectable features for different applications. The firmware images of all *Brownie* devices – hubs, leaf nodes for different sensors/actors – and all microcontroller models are created from the same source code.

The *Home2L* build system creates a number of different firmware images, which is referred to as the *Brownie Family*. Examples are:

ahub.t85: Root (A-type) hub for *ATtiny85* devices.

bhub.t85: Secondary (B-type) hub for *ATtiny85* devices.

mat4x8.t861: Sensor device driving a diode matrix of 4 rows and 8 columns, suitable to read up to 32 sensor switches (for *ATtiny861* devices).

win.t84: For windows with electrical shades: Can drive a shades actuator and two push buttons. Additionally supports a temperature sensor.

win2.t84: For windows with electrical shades: Can drive actuators and buttons for two shades (or electrical window opener(s)).

init.t84, init.t85, init.t861: Maintenance systems and initialization data for preparing new devices (see Sections 7.4.2 and 7.9).

Custom firmware images can be added to the family by editing the file `brownies/avr/Family.mk`. Table 7.3 lists the available features. The first column lists the feature name as used in this document and by `home2l-brownie2l`. The second column shows the names of the compiler parameter (preprocessor definitions) to be defined in `brownies/avr/Family.mk` to activate and configure the respective features. Details on these parameters can be found in the code documentation (Brownies / Feature Selection). Explanations on the more sophisticated feature modules is given in Section 7.7.

(For experts) A brief guide on how to adds new feature modules to support new sensors or actors can be found in the code documentation (Brownies / API).

7.4.2. Maintenance and Operational System

To allow firmware updates and recovery in the field, the program memory (flash) contains two firmware instances: The *operational* system is the main firmware. The *maintenance* system is a minimal firmware containing just the TWI slave communication stack and all functionality for accessing the configuration record and for flash programming. The maintenance system allows to upload and install the operational firmware and vice versa: The maintenance firmware can be updated from a running operational system.

Feature	Source (*Family.mk*)	Description
gpio	GPIO_IN_PRESENCE, GPIO_OUT_PRESENCE, GPIO_IN_PULLUP, GPIO_OUT_PRESET	General-Purpose Input Output (GPIO). Thy can statically be assigned as inputs or outputs. For outputs, the initial value can be defined. For inputs, the internal pull-up can be activated or not.
maintenance	IS_MAINTENANCE	Informal bit to indicate if the firmware is a maintenance firmware.
timer	WITH_TIMER	Ticks timer (typically enabled automatically if some other features requires the timer)
notify	TWI_SL_NOTIFY	Device may issue host notifications.
twihub	WITH_TWIHUB, TWI_HUB_*	TWI Hub functionality.
matrix($r \times c$)	MATRIX_ROWS (> 0), MATRIX_COLS (> 0), MATRIX_*	A sensor matrix with r rows and c columns is enabled. As outputs (rows) and inputs (columns), GPIO ports are used. These cannot be used by the *gpio* feature.
temperature	WITH_TEMP_ZACWIRE, TEMP_*	*ZACwire*-based temperature sensor (e.g. *TSIC 206/306*)
adc_0	ADC_PORTS (≥ 1)	(First) analog input (#0)
adc_1	ADC_PORTS (≥ 2)	Second analog input (#1)
shades_0	SHADES_PORTS (≥ 1), SHADES_*, SHADES_0_*	(First) shades or window actuator (#0)
shades_1	SHADES_PORTS (≥ 2), SHADES_*, SHADES_1_*	Second shades or window actuator (#1)

Table 7.3.: List of *Brownie* device features

The program (flash) memory layout is shown in the first row of Table 7.4. Switching between the two systems (operational and maintenance) is done by rewriting the reset and interrupt vector table (addresses 0x0000 – 0x003f). Hence, the system selection is persistent, and each system can operate without any dependencies from the other.

For the case that a new operational system does not work properly, a *resurrection* routine is implemented in each operational system: If the device is reset and both the TWI SCL and SDA lines are low for at least 250 ms, the system automatically activates and boots the maintenance system.

All procedures related to (re-)booting, activating a maintenance or operational system, firmware installation and resurrection are supported by `home2l-brownie2l`, which offers handy commands for them.

7.4.3. Virtual Memory Layout

Table 7.4 shows the layout of the virtual memory of *Brownie* devices as exposed by the *memory read/write* operations of the *Home2L* bus protocol.

The program flash memory is exposed completely to allow software updates. The SRAM contents are exposed for debugging purposes only. The EEPROM section exposes the *ID record* and the *configuration record* (see Section 7.4.4).

The *version and feature ROM (VROM)* is a special read-only area in which the *feature record* is exposed. It contains all information relevant to identify the firmware: The firmware image name, the build version and the device features (see Section 7.4.1 and Table 7.3). Details can be found in the code documentation (`SBrFeatureRecord`).

Address	Description
0x0000 – 0x7fff	Program memory (flash)
0x0000 – 0x003f	Reset and interrupt vectors
0x0040 – 0x09ff	Maintenance system
0x0a00 – 0x1fff	Operational (main) system
0x8000 – 0x8fff	SRAM contents
0x9000 – 0x9fff	EEPROM contents (ID and configuration)
0xa000 – 0xafff	Version and feature ROM (VROM)

Table 7.4.: Virtual memory layout

7.4.4. Configuration Record

The *configuration record* stores a number of parameters that can be modified at run time. These are on the one hand general parameters like the device's TWI address and, on the other hand,

feature-specific settings such as the subnet specification for hubs or calibration parameters for shades actuators.

A list of the configuration variables is shown in Table 7.5. The tool `home2l-brownie2l` maintains an up-to-date list and online help for all configuration variable. Details on the data structures can be found in the code documentation (`SBrConfigRecord`).

7.4.5. Register Map

The map of 64 registers exposed by the *Home2L* bus protocol is documented in code documentation (Brownies / Interface / Registers). Also, `home2l-brownie2l` maintains an up-to-date register list in its online help.

7.4.6. Pin Assignments

The pin assignments of the *ATtiny85*, *ATtiny84*, and *ATtiny861* devices and their subtypes (e.g. *ATtiny84a*) are shown in Figures 7.4, 7.5, and 7.6, respectively.

Information on changing the assignments or adding other microcontrollers can be found in the code documentation (Brownies / Pins).

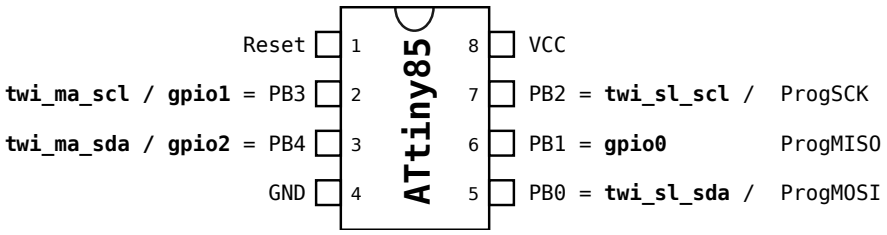


Figure 7.4.: *ATtiny85* pinout

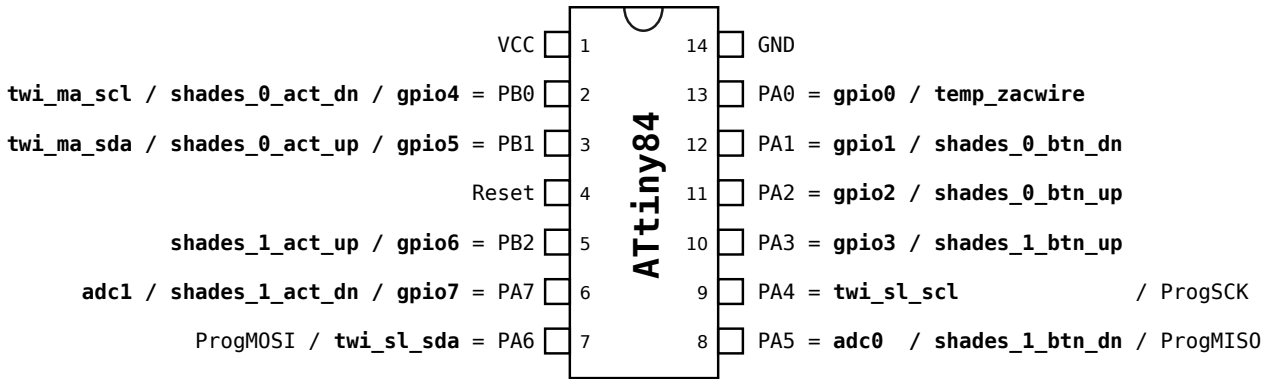
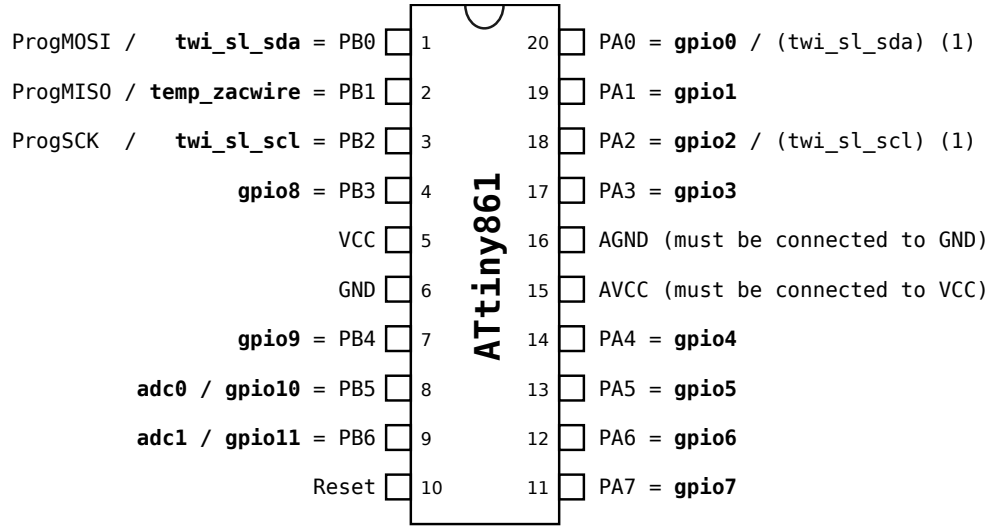


Figure 7.5.: *ATtiny84* pinout



(1) alternative USI location, see USIPP / USIPOS (Manual sec. 13.5.5)

Figure 7.6.: *ATtiny861* pinout

7.5. The Maintenance Tool: `home2l-brownie2l`

The *Brownie maintenance tool* (`home2l-brownie2l`) is a powerful program to set up, configure or diagnose *Brownies*. With the tool you can:

- show the up-to-date register map and list of configuration options,
- scan a bus tree for available devices,
- check a complete bus tree against the database (`brownies.conf`, see Section 7.6),
- read or write *Brownie* registers,
- read and dump *Brownie* memory,
- configure a *Brownie* and read out its current configuration,
- measure and calibrate the internal timer (the *AVR OSCCAL* register),
- program the flash memory from firmware image (ELF) files,
- reboot a device,
- activate a different firmware (maintenance or operational),
- perform hub-related maintenance tasks like powering on/off a sub tree or resurrecting a sub tree,
- run a communication test, printing performance and error statistics,
- collect and show link statistics,
- do simple batch processing, e. g. to do firmware and/or configuration upgrades on a complete tree.

Details about the respective operations are provided by the online help ("help" command).

The *Brownie tool* can operate without any database and thus help to create or update a database. If available, the database is specified with the `br.database` parameter.

The *Brownie tool* may access the bus either directly or attach to a running driver (`home2l-drv-brownies`). The interface is specified by the `br.link` parameter. Attaching to a running driver allows to keep the driver running for a long time and collect long-term statistics. While attached, the driver suspends its normal operation (resources get invalidated during that time) to avoid collisions with actions performed by the *Brownie tool*.

7.6. The Brownie Database (`brownies.conf`)

The *Brownie database* represents the plan of a *Brownie* bus tree. It is a text file with one line per *Brownie*, which is a space-separated list of parameter assignments. An example is available at `tools/etc/brownies.conf`. Table 7.5 lists the possible parameters. The first block of parameters are information used by the maintenance tool `home2l-brownie2l` and the driver `home2l-drv-brownies`. The remaining parameters are configuration options for the *Brownie* firmware.

Parameter	Description
fw	Firmware image name
version	Version number
features	Feature code
id	Brownie ID string
adr	Brownie TWI address
osccal	Timer calibration: OSCCAL register
hub_maxadr	(hub) Hub subnet end address
hub_speed	(hub) Hub and TWI master delay factor, a value of n adds a delay of approximately n*10µs per bit
sha0_du	(shades) Shades #0 calibration: up delay [s]
sha0_dd	(shades) Shades #0 calibration: down delay [s]
sha0_tu	(shades) Shades #0 calibration: total time to move up [s]
sha0_td	(shades) Shades #0 calibration: total time to move down [s]
sha1_du	(shades) Shades #1 calibration: up delay [s]
sha1_dd	(shades) Shades #1 calibration: down delay [s]
sha1_tu	(shades) Shades #1 calibration: total time to move up [s]
sha1_td	(shades) Shades #1 calibration: total time to move down [s]

Table 7.5.: List of *Brownie* database (upper part) and configuration (lower part) parameters

The database can either be written manually, created semi-automatically from already configured *Brownies*, or first be written manually and later be updated semi-automatically. The maintenance tool (`home2l-brownie2l`) does not require a database for any of its tasks. Its `config` and `program` commands can read their parameters either from the command line, but also from the database, if a corresponding entry exists. After (re-)configuring a *Brownie* tree manually, the command `scan -d` reads out all connected *Brownies* and outputs database entries.

A good practice to create the database is:

1. Plan the *Brownie* bus tree and write down the plan into a simple database file containing only the *"adr"*, *"id"*, and *"fw"* parameters. For hubs, *"hub_maxadr"* should be defined, too.
2. Initialize all devices (see Section 7.9), program their operational firmware (`program -d <ID>`) and configure them according the preliminary database (`config -d <ID>`).
3. Use `home2l-brownie2l` to tune their parameters during real operation.
4. Read out all actual parameters (`scan -d`) and copy them into the database for later reconfiguration.

The *Brownies driver* `home2l-drv-brownies` requires the database. Devices without a valid and consistent database entry are ignored by the driver. The `scan -v` command of `home2l-brownie2l` can be used to validate the database against the real hardware.

A *Brownie* database always refers to one bus tree with a single root master. If there are multiple trees, a separate database file must be created for each tree. If multiple trees are connected to the same Linux machine via multiple *i2c* interfaces, a separate *Home2L* instance must be running for each interface.

7.7. Brownie Device Features

This section gives background information on the more complex device features (*firmware modules*). The details can generally be found

- in the code documentation (Brownies / Features) for compile-time options and technical aspects,
- in Section 10.4.2 for run-time options,
- in Section 10.4.3 for the exported resources.

7.7.1. Temperature Sensor (*temperature*)

The *temperature* module presently supports temperature sensors using the single-wire *ZACwire* protocol. Supported sensors are *TSic 206/306* devices, which are claimed to not need calibration, have high accuracy and low power consumption.

The sensors send two-byte datagrams with the current temperature value at a rate of 10 per second. The *ZACwire* protocol uses simple pulse-width modulation and a bit rate of 8 kBit/s.

In the *Brownie* firmware, the *ZACwire* datagrams are received by a software routine with active waiting loops. During the receipt of a message (each 100 ms), interrupts are disabled for approximately 2.5 ms, during which no other activity is possible. At the time of writing, the firmware does not contain any other routines with real-time requirements in this order of magnitude, so that this is not an issue.

7.7.2. Switch Matrix (*matrix*)

Purpose of the *matrix* module is to drive a larger number of sensors arranged as a diode matrix with special options for window sensors.

The matrix has a preconfigured number of *rows R* and *columns C*. The last R GPIO pins are assigned to rows and act as stimulation outputs, the C GPIO pins preceding the row lines are the sensing inputs and assigned to columns. Periodically, the firmware drives a high value (VCC) to exactly one row output (the others are kept at GND level) and reads the column inputs. The column inputs must have pull-down resistors towards GND. This way, the firmware can detect which column signals have a connection to the row signal. The sensor switches must be installed such that each of them connects a unique combination of row and column lines in line with a diode from the row line towards the column line.

The sensor values are debounced internally, the scan frequency can be adjusted with the `MATRIX_T_PERIOD` setting. An internal event queue ensures that quick changes and the order of changes can be preserved even in presence of delays in the communication with the Linux host.

In addition, the *matrix* module can combine the information of two window reed sensors into a single resource indicating whether a window is closed, open or tilted. Different placements of reed sensors are supported. Details can be found in the documentation of `br.matrix.win.<brownieID>.<winID>`.

7.7.3. Window Shades and Actuators (*shades*)

The *shades* module implements the functionality of one or two shades or window opening actuators together with a pair of push buttons (up/down) each.

The state of an actuator is represented by the resource `brownies/<brownieID>/shades<n>/pos`, which is a percentage value between 0% (= "shades up" or "window closed") and 100% (= "shades down" or "window open"). Movement can be triggered by setting a request with the target position to this resource. If the user pushes one of the two buttons, an automatic request with the desired position is generated.

The current position is traced with the help of a timer. To make it work, the module must be calibrated by setting the *"sha_*"* configuration options (see Section 7.4.4). These are four time constants to be supplied:

- the delay after the actuator is powered on until the shades start moving up/down (two values, typically between 0.2 and 1.0 seconds),
- the total time to move all way up or down, respectively (two values).

The actuators must have internal limit switches to avoid damage.

In case of a bus failure, the actuators remain controllable by the buttons directly. In addition, they can be configured to automatically move into a fail-safe position (e.g. "window closed") in case of a bus failure. This behavior is compiled into the firmware by the `SHADES_0_RINT_FAILSAFE` /

SHADES_1_RINT_FAILSAFE feature settings and cannot be (accidentally!) disabled or changed by configuration.

The actuators are controlled in a way that they are never driven into both directions simultaneously and that quick direction changes are avoided (see the SHADES_REVERSE_DELAY feature setting).

7.7.4. Analog Sensing (*adc*)

The ADC feature is not implemented yet. This section is under construction.

7.8. Circuit Examples

Disclaimer

Building electronic circuits requires adequate knowledge in electronics. Modifying the electrical installation of a building is inherently dangerous and may result in serious damage, injury or even death if not done properly.

You expressively agree to hold the authors of the Home2L suite and of this document harmless for any property damage, personal injury and/or death, or any other loss or damage that may result from your use of the information or software provided.

This material is provided "as is". Use it wisely, it is at your own risk!

7.8.1. General Considerations

Depending on the application, subsets the microcontroller port pins are used as inputs, as inputs with internal pull-up or as outputs, respectively. The following points concerning the pin configuration must be taken into account during circuit design:

a) Each *operational* firmware sets all user port directions and eventually activates internal pull-ups as implied by the respective device features of the firmware. This happens very early during startup.

b) Ports not used by the *operational* firmware are generally configured as inputs with the internal pull-up activated. Hence, they can left open in the circuit without problems. It is allowed to connect them to VCC, but they should *not* be connected to GND since otherwise unnecessary current draw and power consumption will occur.

c) A *maintanance* firmware puts *all* user pins in a high-impedance state with the internal pull-ups *deactivated*. This is consistent with the state immediately after power-on (details can be found in the microcontroller documentation).

   User pins in this context are all pins except *Reset*, *VCC*, *GND* and the TWI slave signals *SCL* and *SDA*.

Hence, for the time a maintenance firmware is running, the user ports are in a high-impedance state, even if they are usually configured as outputs or inputs with internal pull-ups. This must be considered in the circuit design: Outputs require appropriate pull-up or pull-down circuitry to avoid unwanted actions (such as the activation of an actuator). Inputs may be floating while the maintenance system is running. This is usually no problem, but may cause additional power consumption, so that the maintenance system should not be activated permanently.

7.8.2. Circuits

The directory `brownies/circuits` contains some sample circuits as inspirations for own designs:

Hub Card (`hubcard`)
: is a board design for one or multiple primary hubs to be placed close to the Linux host computer.

  The power supply of the sub tree can be switched on or off, so that the hub can power-cycle and reset the tree. This is also a prerequisite to use the resurrection functionality for the sub tree devices.

  The power supply is +5V and the same voltage is expected at the slave TWI interface. To connect the hub(s) to a mini computer with a lower *i2c* voltage, a level shifter can be used. The NXP Application Note AN10441 contains useful information on this, *BSS138* is a suitable transistor model.

Matrix 4x8 (`matrix4x8`)
: is board incorporating a *Brownie* capable of monitoring up to 32 switch sensors (e.g. window reed contacts). In addition, the board can host a B-type hub and a temperature sensor.

Window (`window`)
: is a small board to be placed near a window. It has two inputs for up/down buttons and two outputs suitable for driving up/down actuator relays. In addition, the board can host a temperature sensor.

Dual Window (`window_dual`)
: is the same as above, but with two pairs of button inputs and relay outputs each.

7.9. Initializing a New *Brownie*

Initializing and integrating a new *Brownie* is done in two stages.

Stage 1: Initializing the Microcontroller

In the first stage, the maintenance firmware, the initial contents of the EEPROM and the fuse bits of the microntroller are set. This requires the initialization image `init.<mcu>.elf`, where `<mcu>` is the microcontroller model and may be `t84`, `t85` or `t861` for an *ATtiny84*, *ATtiny85* or *ATtiny861* device, respectively.

This step can be performed with any programming facility of your choice. For example, with *avrdude(1)* and an *AVRISP mkII* programmer, the command to initialize a *t85* device is:

```
$ export MCU=t85; avrdude -c avrisp2 -p $MCU \
   -U  hfuse:w:init.${MCU}.elf \
   -U  efuse:w:init.${MCU}.elf \
   -U eeprom:w:init.${MCU}.elf \
   -U  flash:w:init.${MCU}.elf
```

This is usually the last time you need your programmer.

Stage 2: Programming the Operational System and Configuration

In the second stage, the *operational* firmware and the configuration record are written to the device. These steps are performed with `home2l-brownie2l` over a *Home2L* or *i2c* bus. The device may either already be installed in the field (which is possible, but not recommended) or connected to a maintenance PC with a local *i2c* adapter.

a) Install the *operational* firmware. It is recommended to have a database entry with the address, the firmware image name and the ID (<*brownieID*>) of the new *Brownie*.

```
brownie2l> open 7
brownie2l> program -d <brownieID>
```

Be sure to *not* try to boot the operational firmware *before* this step (*"boot -o"* command), since this would result in an unresponsive device and probably force you to repeat stage 1.

b) Write the configuration record containing, among others, the TWI address and the *Brownie* ID:

```
brownie2l> config -d <brownieID>
```

c) Boot the device into the operational system:

```
brownie2l> boot -o
```

d) Check its new identity:

```
brownie2l> open
```

e) Optionally, perform a communication test:

```
brownie2l> test
```

8. *WallClock* – An Unobtrusive GUI

8.1. Overview

The *WallClock* is intended to be used as what the name suggests – as a wall clock, but with some extra functionality in an unobtrusive way.

The main display (see Figure 8.1) shows the time and date, local weather information (if the driver `home2l-drv-weather` is set up) and some status about the building (see Section 2.3.2 for more explanations).

Figure 8.1.: The *WallClock* Main Screen

The following functional modules (referred to as *applets*), which are frequently useful in a private household, are integrated in the *WallClock*:

- a SIP-based video phone client (for door phone and inter-room communication),
- a calendar,
- a music player (*MPD* client) designed for a distributed setup with many rooms and muktiple user in a household,
- an alarm clock to use the *WallClock* device as a radio alarm clock.

Ideally, a *WallClock* device is installed in each room in the house. Due to its resource-aware implementation, low-cost hardware such as cheap or older second-hand Android tablets is sufficient.

The *WallClock* is implemented in native code (C/C++) and uses *SDL2* for its UI. This makes it portable and efficient. So far, the *WallClock* has been tested on (PC) Linux and on Android, but ports of *SDL2* to many other environments exist (see `https://libsdl.org`).

8.2. The Floor Plan

8.2.1. Overview

The floor plan applet (see Figure 8.2) allows to view and to interact with all kinds of gadgets in an intuitive way. Possible gadgets include windows, door locks, garage doors, lights, mailboxes, phones, and motion detectors. Intercom phone calls can be initiated by pushing the respective phone icon.

Depending on the presence state, gadgets can be highlighted to attract the user's attention, if, for example, a window is still open at night or the main door not locked.

The home screen of the *WallClock* shows a mini floor plan, which also shows the states of gadgets (see Figure 8.3).

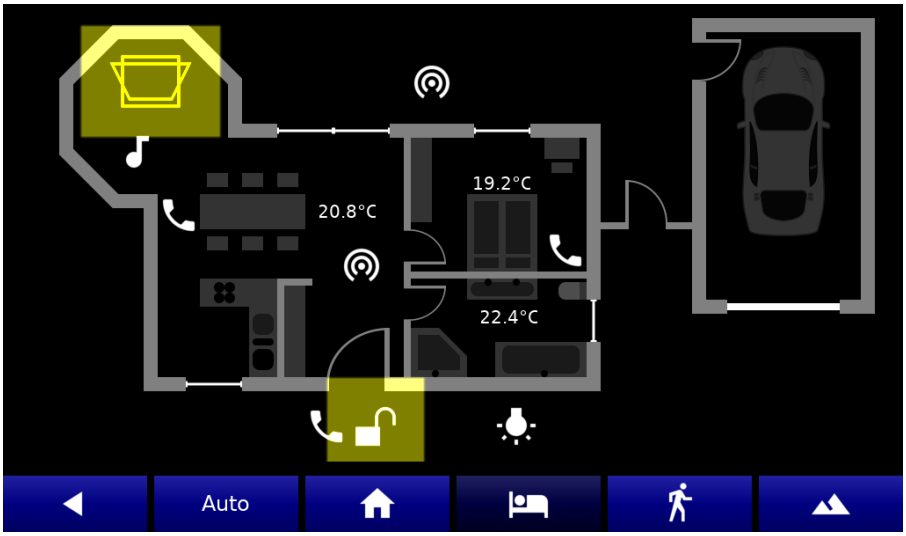


Figure 8.2.: The *WallClock* floor plan

8.2.2. Creating a Custom Floor Plan

The floor plan is specified by an SVG drawing and is editable using Inkscape, a powerful free drawing tool. A symbol library contains all available gadgets. They can be placed in the drawing, which besides this can be drawn freely. The *Home2L Floor Plan Compiler* (`home2l-fpc`) analyses the drawing and creates all necessary data files for the *WallClock* application as well as a template

Figure 8.3.: Mini floor plan on the home screen warning about two open windows

file listing all resources to be used as a template for the `resources.conf` file. Gadgets and their resources are identified by their *object IDs* in the SVG drawing.

Step 1: Draw the floor plan using *Inkscape*.

- The file `wallclock/floorplan-template.svg` should be used as a template. It also contains a number of hints on how to draw it. After a new installation, a copy of this file can usually be found as `$HOME2L_ROOT/etc/floorplan.svg`. If the file is not at this location: Copy it there.
- Gadgets are available as a symbol library, which is embedded in the template SVG file.
- The size of the drawing can be 128x64 or 256x128 pixels, depending on the space required for you home.
- The template has a set of fixed layers. They must not be renamed, reordered or removed, but serve certain purposes:

  Layer 'helper' is ignored during compilation and will be invisible. It may contain helper material such as a blueprint of the building.

  Layer 'building' will be rendered as drawn. The contents are arbitrary. However, for styling and readability reasons, only grey-level colors should be used and all objects should be aligned to pixel boundaries.

  Layer 'gadgets' contains all gadgets. Refer to the documentation in the file for how they may be scaled and rotated.

  Layer 'zoom' is reserved for future use.

Step 2: Set meaningful IDs for all your gadgets.

This can be done using the "Object Properties" dialog in *Inkscape*. IDs should be lowercase and may contain "_" and "-" characters. You are free to select your IDs, but it is recommended to use a naming scheme like `<room>-<gadget>-<subID>`, for example: `kitchen-shades` or `dining-window-l`.

Step 3: Compile your SVG file using the floor plan compiler.

```
$ home2l fpc floorplan.svg
```

This will create a directory *floorplan.fpo* with a number of files directly readable by the *WallClock* at run-time. One file, *sample-resources.conf* is a template to be used in the next step. It contains a list of all resources referred to by the floor plan.

Step 4: Update your `resources.conf` file to assign resources to the floor plan gadgets.

Copy the contents of `floorplan.fpo/sample-resources.conf` into your `resources.conf` file and adapt it there. The most relevant part is a list of alias definitions. The alias names are generated from your object IDs. The alias targets must be adapted to point to the real resources.

As an aid for testing, a set of local signals are defined and the aliases point to them, so that it is possible to test the floor plan in the UI without functionality.

Step 5: Copy the floor plan object into the 'etc' directory.

To activate the floor plan, make sure that the floor plan compiler (`home2l-fpc`) output resides in `$HOME2L_ROOT/etc/floorplan.fpo`.

8.3. The Phone

The *Phone* applet implements a SIP-based VoIP phone with video fumctionality and the ability to act as a door phone in conjunction with `home2l-doorman`. For example, the applet can offer a door opener button if its peer is a door phone. Figures 8.4 and 8.5 show the phone while ringing and during a video phone call, both in door phone mode.

Together with a private branch exchange (PBX) software such as *Asterisk*, the *WallClocks* can be used as an intercom system for the house and for a sophisticated door phone system with multiple door bells and multiple answering stations in the building.

In order to compile the *WallClock* with phone capabilities, *PJSIP* or *Liblinphone* is required.

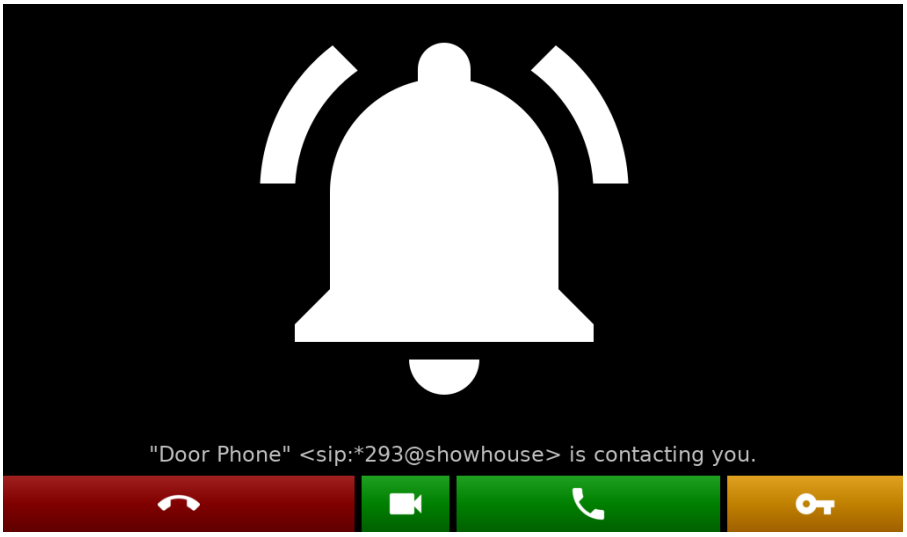


Figure 8.4.: The *WallClock* phone: Ringing in door phone mode

Figure 8.5.: The *WallClock* phone: Door phone with video[1]

[1]The video image has been created using: The Beatles magical mystery tour, Parlophone Music Sweden, CC BY 3.0

8.4. The Calendar

The *Calendar* applet is a graphical calender tool supporting locally and remotely stored calendars. It supports multiple independent calendars in a single view, allowing to distinguish private from business appointments or to manage the activities of multiple family members (see Figure 8.6).

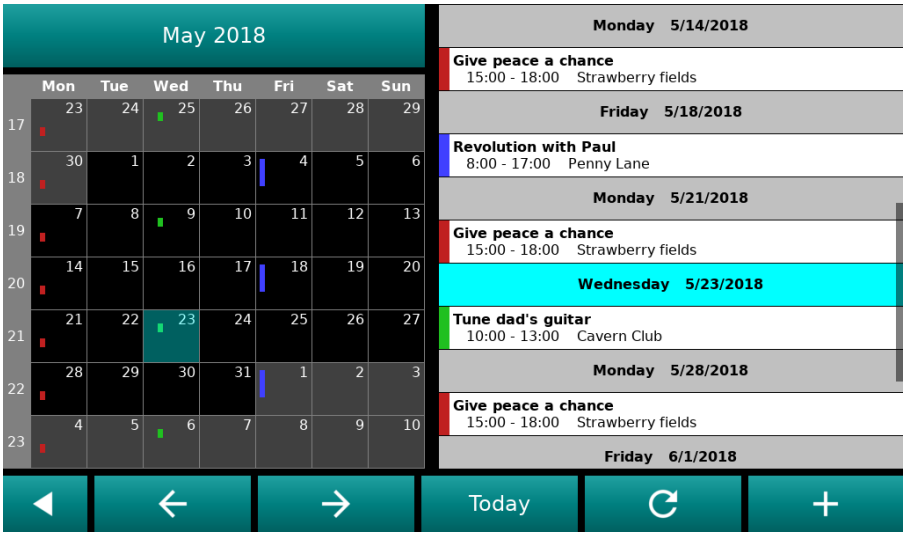


Figure 8.6.: The *WallClock* calendar view

As its backend and storage format, *remind(1)* is used. *remind(1)* is a mature and powerful command line tool supporting single events with and without times as well as many sorts of recurring events. The main advantage is the fact that a complete calendar is maintained in one *reminders* file with one event per line. The file is human-readable and can easily be edited by hand. Synchonization, merging and even revision control can easily be accomplished by standard tools such as *diff(1)*, *meld(1)* and *git(1)*. Several frontends exist for *remind* (*tkremind(1)*, for example), the *WallClock Calendar* is a new one with special support for multiple calendars.

The calendar files can be stored wherever the user likes – locally or on a home server. For any editing operation, *WallClock Calender* generates a *patch(1)* fragment and passes it to *patch* on the machine storing the calendar file.

This method for accessing calendar files allows a high degree of flexibility for where and how they are stored, and the use of *patch(1)* supports concurrent editing operations from multiple *WallClock* instances.

To use the *Calendar* applet, the *remind(1)* command line tool must be installed on some computer reachable by *ssh(1)* by the *WallClock*.

8.5. The Music Player

The *WallClock Music Player* is a front end for the *MPD* music player daemon (https://www.musicpd.org). It aims to support a home installations with multiple rooms, multiple users, and

multiple virtual or physical stereo systems. In any room, any user shall be able to control any stereo system using any *WallClock* device. Everywhere in the house, he shall have access to the complete music collection and be able to get it transformed into acoustic air waves by any device he likes: hifi speakers in the living room, other speakers in the kitchen, earphones connected to the local device, or bluetooth speakers coupled with the *WallClock* device. Of course, a user can switch between them anytime and "take" the currently playing music with him as he moves to another room.

Technically, a "virtual stereo system" is an installed *MPD* instance running on some computer in the household. It has to be declared in `home2l.conf` using `music.<MPD>.host` and related parameters.

If the *MPDs* are configured with an http streaming output, the music can be streamed back and played on the local device (*WallClock* must be compiled with *GStreamer* support for this).

Figure 8.7 shows a screenshot with the usual *MPD* controls on the left and a database and playlist browser on the right. The title bar can be pushed to navigate up or switch between the local collection and playlists.

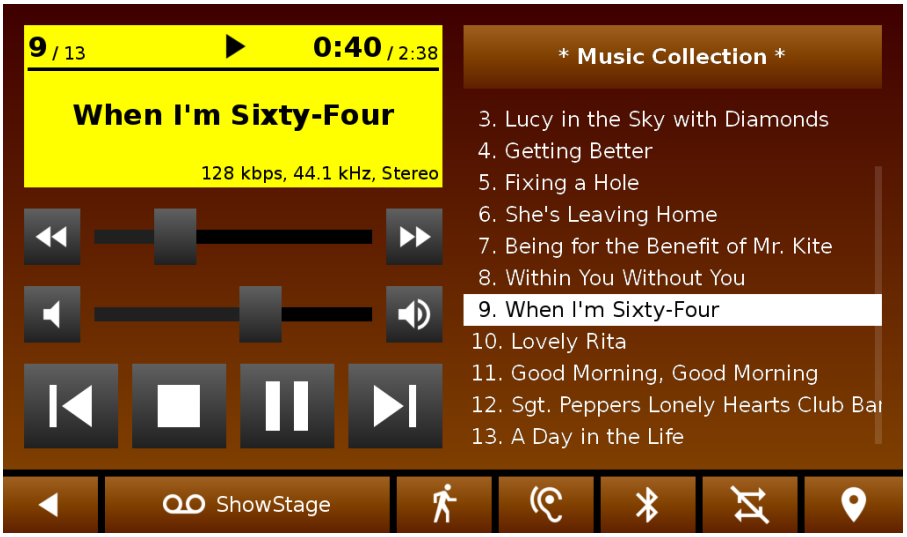


Figure 8.7.: The *WallClock* Music Player

With the buttons in the bottom line you can (from left to right)

- select the "virtual stereo system" (*MPD* instance).
- take the currently playing music from one *MPD* instance to another (e.g. to seamlessly continue listening a song when moving from one room to another).
- select the output device (any output configured with *MPD* or stream to the local device, if available).
- enable or disable Bluetooth (Android).
- select the repeat mode.
- navigate to the currently playing song.

The `ui/mute` resource allows to mute the music player by automation rules, for example, if the phone rings.

A long push on the "Music" launcher on the main screen (Figure 8.1) turns the music player on or off without switching to the music screen.

8.6. The Alarm Clock

The alarm clock can be enabled by pushing the clock display on the main screen. For each weekday, an individual alarm time can be programmed.

By default, the music player is activated on alarm. If the music player fails or the music is not loud enough (see `alarm.minLevelDb`), a ringing sound is played as a fallback.

For the case that the alarm clock fails completely, a dead-man script can be submitted to another computer, which can then wake you up with some other means (e.g. by a wakeup call to your mobile phone). See `alarm.extAlarmHost`, `alarm.extAlarmCmd` and `alarm.extAlarmDelay` for details.

8.7. List of Configuration Parameters

8.7.1. Parameters of Domain `ui`

`ui.passiveBehaviour` (bool) [= false] `wallclock/system.C:41`

Main application bahaviour.

Set the general application behaviour. If set to false (default), the app tries to control the display brightness, has own mechanisms for dimming the screen and tries to auto-activate after some time when the user shows no activity in another app (launcher-like behaviour).

If set to true, most of these mechanisms are disabled, and the app behaves like a normal app. All settings are controlled by the (Android) system.

`ui.standbyDelay` (int) [= 60000] `wallclock/system.C:52`

Time (ms) until standby mode is entered.

`ui.offDelay` (int) [= 3600000] `wallclock/system.C:55`

Time (ms) until the screen is switched off.

`ui.lightSensor.minLux` (float) [= 7] `wallclock/system.C:661`

Any Lux value below this will be rounded to this.

`ui.lightSensor.alOffset` (float) [= 20] `wallclock/system.C:664`

Linear part of the "apparent light" function (in Lux).

`ui.lightSensor.alFilterWeight` (float) [= 0.1] `wallclock/system.C:667`

Apparent light filter factor.

`ui.lightSensor.acThreshold` (float) [= 0.02] `wallclock/system.C:670`

Apparent change threshold to report a change.

The app tries to detect the presence of people in the room by monitoring the light sensor and waking up the app on a quick change in light. This and the previous parameters can be used to tune the sensitivity of this wakeup mechanism. Please refer to the source code to understand the exact algorithm.

Please do not expect too much – typical light sensors are not well suited for presence detection. If unsure, leave these settings with their defaults.

`ui.display.minLux` (float) [= 10.0] `wallclock/system.C:683`

Reference Lux value for the "minimum" brightness values.

`ui.display.typLux` (float) [= 100.0] `wallclock/system.C:686`

Reference Lux value for the "typical" brightness values.

`ui.display.maxLux` (float) [= 1000.0] `wallclock/system.C:689`

Reference Lux value for the "maximum" brightness values.

THe display brightness is adjusted according to a two-piece piece-wise linear function depending on the logarithm of the Lux value. This and the previous parameters define the Lux values for the three supporting points of the piece-wise linear function.

`ui.display.activeMin` (float) [= 0.5] `wallclock/system.C:699`

Minimum display brightness in active mode (percent).

`ui.display.activeTyp` (float) [= 0.7] `wallclock/system.C:702`

Typical display brightness in active mode (percent).

`ui.display.activeMax` (float) [= 1.0] `wallclock/system.C:705`

Maximum display brightness in active mode (percent).

`ui.display.standbyMin` (float) [= 0.25] `wallclock/system.C:709`

Minimum display brightness in standby mode (percent).

`ui.display.standbyTyp` (float) [= 0.35] `wallclock/system.C:712`

Typical display brightness in standby mode (percent).

`ui.display.standbyMax` (float) [= 0.5] `wallclock/system.C:715`

Maximum display brightness in standby mode (percent).

`ui.longPushTime` (int) [= 500] `wallclock/ui_base.C:51`

Time for a long push in milliseconds.

`ui.longPushTolerance` (int) [= 16] `wallclock/ui_base.C:54`

Tolerance for motion during a long push in pixels.

`ui.resizable` (bool) [= true] `wallclock/ui_base.C:57`

Selects whether the UI window is resizable on startup.

This determines the X window's "resizable" option when the application starts. Under a tiling window manager (e.g. awesome), this also determines whether the window opens as a floating window (false) or in tiling mode (true).

The flag can be toggled at runtime by pushing the F12 button.

`ui.audioDev` (string) `wallclock/ui_base.C:66`

Audio device for UI signaling (phone ringing, alarm clock).

`ui.sysinfoCmd` (string) [= "bin/h2l-sysinfo.sh"] `wallclock/app_home.C:57`

Name of the sytem information script.

This script is executed repeatedly and its out displayed when the user opens the "about" screen.

`ui.sysinfoHost` (string) `wallclock/app_home.C:64`

Host on which the system information script is executed.

If set, the system information script is executed on the given remote host.

`ui.accessPointRc` (string) [= "/local/accessPoint"] `wallclock/app_home.C:70`

Resource (boolean) for local (wifi) access point status display.

`ui.bluetoothRc` (string) [= "/local/ui/bluetooth"] `wallclock/app_home.C:73`

Resource (boolean) for local bluetooth status display.

`ui.outdoorTempRc` (string) [= "/alias/weather/temp"] `wallclock/app_home.C:77`

Resource (temp) representing the outside temperature for the right info area (outdoor).

`ui.outdoorData1Rc` (string) [= "/alias/ui/outdoorData1"] `wallclock/app_home.C:80`

Resource for the upper data field of the right info area (outdoor).

`ui.outdoorData2Rc` (string) [= "/alias/ui/outdoorData2"] `wallclock/app_home.C:83`

Resource for the lower data field of the right info area (outdoor).

`ui.indoorTempRc` (string) `wallclock/app_home.C:87`

Resource (temp) representing the outside temperature for the right info area (indoor).

`ui.indoorData1Rc` (string) [= "/alias/ui/indoorData1"] `wallclock/app_home.C:90`

Resource for the upper data field of the right info area (indoor).

`ui.indoorData2Rc` (string) [= "/alias/ui/indoorData2"] `wallclock/app_home.C:93`

Resource for the lower data field of the right info area (indoor).

`ui.radarEyeRc` (string) [= "/alias/weather/radarEye"] `wallclock/app_home.C:97`

Resource for the radar eye as provided by the 'home2l-weather' driver.

`ui.lockSensor1Rc` (string) [= "/alias/ui/lockSensor1"] `wallclock/app_home.C:101`

Resource (bool) for the first lock display in the left info area (building).

`ui.lockSensor2Rc` (string) [= "/alias/ui/lockSensor2"] `wallclock/app_home.C:104`

Resource (bool) for the first lock display in the left info area (building).

`ui.motionDetectorRc` (string) [= "/alias/ui/motionDetector"] `wallclock/app_home.C:107`

Resource (bool) for the motion detector display in the left info area (building).

`ui.motionDetectorRetention` (int) [= 300000] `wallclock/app_home.C:110`

Retention time (ms) of the motion detector display (OBSOLETE).

`ui.radarEye.host` (string) `wallclock/app_home.C:117`

Host to run 'ui.radarEye.cmd' on. (OBSOLETE: Use string from URI "/alias/ui/radarEye" instead).

`ui.radarEye.cmd` (string) [= "cat $HOME2L_ROOT/tmp/weather/radarEye.pgm"]
`wallclock/app_home.C:120`

Command to obtain .pgm file for the radar eye. (OBSOLETE: Use string from URI "/alias/ui/radarEye" instead).

`ui.launchMail` (string) `wallclock/app_home.C:429`

Android intent to launch a mail program (optional, Android only).

Only if set, a launch icon is shown on the home screen.

`ui.launchWeb` (string) `wallclock/app_home.C:434`

Android intent to launch a web browser (optional, Android only).

Only if set, a launch icon is shown on the home screen.

`ui.launchDesktop` (string) [= false] `wallclock/app_home.C:439`

If true, the home screen gets an icon to launch the Android desktop (Android only).

`ui.launchWeather` (string) `wallclock/app_home.C:442`

Android intent to launch a weather app (optional, Android only)..

If set, it will be launched if the weather area or radar eye are pushed.

8.7.2. Parameters of Domain `floorplan`

`floorplan.rcTree` (string) [= "/alias"] `wallclock/floorplan.C:42`

Root of the resource tree for floorplan gadgets.

Resources for floorplan gadgets are expected to have an ID like `<floorplan.rcTree>/<floorplan ID>/<gadget ID>/<resource>`.

`floorplan.useStateRc` (string) [= "/local/timer/twilight/day"] `wallclock/floorplan.C:49`

Resource representing the current home's use state (present, absent, ...).

If defined, certain sensor data is highlighted (e.g. open windows or locks) are highlighted depending on the use state.

The resource may have type 'rctUseState' or 'bool'. A boolean value of 'false' is interpreted as 'night', a boolean value of 'true' is equivalent to 'day'.

floorplan.weatherRc (string) wallclock/floorplan.C:60

Resource representing the weather status.

If defined, certain sensor data is highlighted depending on the weather.

At present, the resource must have type 'rctBool', and a value of 'true' is interpreted as a warning of any kind (rain or worse). In the future, an enumeration type may be introduced here to distinguish different warning conditions (e.g. storm, rain, snow).

floorplan.requestAttrs (string) wallclock/floorplan.C:72

Request attributes for user interactions with the floorplan [rc.userReqAttrs].

Define request attributes for any user interactions with the floorplan.

By default, the value of rc.userReqAttrs is used.

floorplan.motionRetention (int) [= 300] wallclock/floorplan.C:81

Retention time (s) for a motion detector display.

floorplan.rwin.shades (bool) [= false] wallclock/floorplan.C:90

Enable/disable the shades resource for roof window (rwin) gadgets..

This sets the default for any floorplan.gadgets.<gadgetID>.shades setting.

floorplan.gadgets.<gadgetID>.shades (bool) wallclock/floorplan.C:96

For roof window (rwin) gadgets: Enable shades resource.

If <gadgetID> refers to a roof window with electric shades, this option should be set 'true' and a resource referred by /alias/<floorplan>/<gadget>/shades is used to control it.

By default, the floorplan.rwin.shades setting is used.

floorplan.rwin.actuator (bool) [= false] wallclock/floorplan.C:107

Enable/disable the actuator resource for roof window (rwin) gadgets..

This sets the default for any floorplan.gadgets.<gadgetID>.actuator setting.

floorplan.gadgets.<gadgetID>.actuator (bool) wallclock/floorplan.C:113

For roof window (rwin) gadgets: Enable an actuator resource.

If <gadgetID> refers to a roof window with an actuator for opening/closing, this option should be set 'true' and a resource referred by /alias/<floorplan>/<gadget>/actuator is used to control it.

By default, the floorplan.rwin.actuator setting is used.

floorplan.gadgets.<gadgetID>.dial (string) wallclock/floorplan.C:124

For phone gadgets: Set the number to dial resource.

Number to dial if a phone icon is pushed in the floorplan. By default, if the gadget ID ends with digits, the trailing digits are dialed with asterix ("*") prepended.

8.7.3. Parameters of Domain alarm

alarm.ringFile (path) [= "share/sounds/alarm-classic.wav"] wallclock/alarmclock.C:36

Audio file to play if the music player fails or for pre-ringing.

alarm.ringGap (int) [= 0] wallclock/alarmclock.C:40

Number of milliseconds to wait before playing the ring file again.

alarm.preRings (int) [= 0] wallclock/alarmclock.C:44

Number of times the ring file is played before the music player is started.

alarm.snoozeMinutes (int) [= 10] wallclock/alarmclock.C:48

Number of snooze minutes.

alarm.tryTime (int) [= 15000] wallclock/alarmclock.C:52

Maximum time in milliseconds to try playing music.

If the music is not playing after this amount of time, the alarm clock reverts to ringing mode.

`alarm.minLevelDb` (int) [= -30] `wallclock/alarmclock.C:59`

Minimum level (DB) required for music.

If the music is below this level (e.g., because a radio station sends silence), the alarm clock reverts to ringing mode.

NOTE: This option only works if the music is output locally using GStreamer.

`alarm.extAlarmHost` (string) `wallclock/alarmclock.C:68`

Host to run an external alarm script on (local if unset).

This can be used to implement a fallback wakeup (e.g. by a wakeup phone call), if the wallclock fails for some reason.

`alarm.extAlarmCmd` (string) `wallclock/alarmclock.C:75`

Command to setup an external alarm.

This can be used to implement a fallback wakeup (e.g. by a wakeup phone call), if the wallclock fails for some reason. The command will be executed as follows:

```
<cmd> -i <hostname> <yyyy>-<mm>-<dd> <hh>:<mm>
```

`alarm.extAlarmDelay` (int) [= 3] `wallclock/alarmclock.C:84`

Delay of the external alarm setting.

Number of minutes n added to the set alarm time before transmitting the request to the external alarm resource.

In case of a failure in the "standby" or "snooze" state, the external alarm will go off n minutes after the time set. In case of a failure during alarming, the external alarm will go off between n and $2n$ minutes after the time set.

8.7.4. Parameters of Domain `var.alarm`

`var.alarm.enable` (bool) [= false] `wallclock/alarmclock.C:101`

Enable the alarm clock as a whole.

`var.alarm.timeSet.<n>` (int) `wallclock/alarmclock.C:105`

Wake up time set for week day $<n>$.

The time is given in minutes after midnight. Week days are numbered from 0 (Mon) to 6 (Sun).

Values < 0 denote that there is no alarm on the respective day. Values < -1 denote a hint to the UI if the alarm on that day is activated: The time is preset by the negated value.

`var.alarm.active` (int) [= 0] `wallclock/alarmclock.C:116`

Presently active alarm time (in minutes after the epoch)..

This variable is automatically set in a persistent way when an alarm goes off and set to 0 when the user switches off the alarm. It is used to recover the ringing state if the app crashes during alarm.

8.7.5. Parameters of Domain `phone`

`phone.enable` (bool) `wallclock/app_phone.C:62`

Enable the phone applet.

`phone.linphonerc` (path) `wallclock/app_phone.C:65`

Linphone RC file (Linphone backend only).

With the Linphone backend, all settings not directly accessible by Home2L settings are made in the (custom) Linphone RC file.

`phone.register` (string) `wallclock/app_phone.C:71`

Phone registration string.

`phone.secret` (string) `wallclock/app_phone.C:74`

Phone registration password.

`phone.ringFile` (path) [= "share/sounds/phone-classic.wav"] `wallclock/app_phone.C:77`

Ring tone file.

`phone.ringGap` (int) [= 2000] `wallclock/app_phone.C:80`

Number of milliseconds to wait between two rings..

`phone.rotation` (int) [= 0] `wallclock/app_phone.C:84`

Phone video rotation.

`phone.doorRegex` (string) `wallclock/app_phone.C:87`

Regex to decide whether a caller is a door phone.

`phone.ringFileDoor` (path) [= "share/sounds/dingdong-classic.wav"] `wallclock/app_phone.C:90`

Ring tone file for door phones calling.

`phone.openerDtmf` (string) `wallclock/app_phone.C:93`

DTMF sequence to send if the opener button is pushed.

`phone.openerRc` (string) `wallclock/app_phone.C:96`

Resource (type 'bool') to activat if the opener button is pushed.

`phone.openerDuration` (int) [= 1000] `wallclock/app_phone.C:99`

Duration of the opener signal.

`phone.openerHangup` (int) [= 0] `wallclock/app_phone.C:102`

Time until the phone hangs up after the opener button is pushed (0 = no auto-hangup).

`phone.fav<n>` (string) `wallclock/app_phone.C:110`

Define Phonebook entry $\#n$ ($n = 0..9$).

An entry has the form "[<display>|]<dial>", where <dial> is the number to be dialed, and (optionally) <display> is the printed name.

8.7.6. Parameters of Domain `calendar`

`calendar.enable` (bool) [= false] `wallclock/app_calendar.C:37`

Enable calendar applet.

`calendar.host` (string) `wallclock/app_calendar.C:40`

Host with calendar files (local if unset).

On the storage host, the tools `GNU patch`, `cat`, and `remind` must be installed. The latter is not needed if `calendar.nearbyHost` is defined.

If a host is set, the application will use `ssh` to run any commands on the host as user `'home2l'`. Hence, to access the calendars as a unified user on the local machine, it is advisable to enter `'localhost'` here.

`calendar.nearbyHost` (string) `wallclock/app_calendar.C:52`

(Optional) Nearby host to accelerate calendar browsing.

If set, calendar files are cached locally (in RAM), and for processing, `remind` is invoked on the machine passed here. This improves the UI responsiveness if the network connection to the storage host or the storage host itself is slow.

On the nearby machine, `remind` must be installed.

`calendar.dir` (string) [= "var/calendars"] `wallclock/app_calendar.C:63`

Storage directory for calendar (reminder) files..

The path may be either absolute or relative to HOME2L_ROOT.

`calendar.<n>.name` (string) `wallclock/app_calendar.C:68`

Name for calendar $\#n$.

`calendar.<n>.color` (int) `wallclock/app_calendar.C:71`

Color for calendar $\#n$.

This should by given as a 6-digit hex number in the form `0x<rr><gg><bb>`.

8.7.7. Parameters of Domain `music`

`music.<MPD>.host` (string) `wallclock/app_music.C:52`

Network host name and optionally port of the given MPD instance..

This variable implicitly declares the server with its symbolic name <MPD>. If no port is given, the default port is assumed.

`music.port` (int) [= 6600] `wallclock/app_music.C:59`

Default port for MPD servers.

`music.<MPD>.password` (string) `wallclock/app_music.C:63`

Password of the MPD instance (optional, NOT IMPLEMENTED YET).

`music.(<MPD>|any)[.<OUTPUT>].name` (string) `wallclock/app_music.C:67`

Define a display name for an MPD server or an output.

`music.streamPort` (int) [= 8000] `wallclock/app_music.C:71`

Default port for HTTP streams coming from MPD servers.

The setting for a particular server <MPD> can be given by variable keyed "music.<MPD>.streamPort".

`music.streamBufferDuration` (int) [= 1000] `wallclock/app_music.C:78`

Buffer length [ms] for HTTP streaming..

`music.volumeGamma` (float) [= 1.0] `wallclock/app_music.C:82`

Gamma value for the volume controller (default and always used for local outputs).

The setting for a particular server <MPD> and optionally output <OUTPUT> can be given by variable keyed "music.<MPD>.[<OUTPUT>].volumeGamma".

`music.streamOutPrefix` (string) [= "stream"] `wallclock/app_music.C:89`

Name prefix for an output.

If the output name has the format "<prefix>[<port>]", it is recogized as a output for HTTP streaming, which can be listened to locally. For concenience, the port number can be appended to the stream prefix.

`music.recordOut` (string) [= "record"] `wallclock/app_music.C:97`

Name for a recording output.

If the output name has this name, it is recogized as an output with recording functionality. Such an output is not listed and selectable by the usual output functionality, but activated if and only if a streaming source is played.

`music.streamDirHint` (string) `wallclock/app_music.C:105`

MPD directory in which radio streams can probably be found.

The "go to current" button navigates to the parent directory of the currently playing song. If the song is not a local file, but a (HTTP) stream, this does not work out of the box. This setting optionally defines the directory to go to, if a non-file is currently played.

`music.recoveryInterval` (int) [= 2000] `wallclock/app_music.C:113`

Retry interval time if something (presently local streaming) fails..

`music.recoveryMaxTime` (int) [= 10000] `wallclock/app_music.C:117`

Maximum time to retry if something (presently local streaming) fails..

`music.autoUnmute` (bool) [= false] `wallclock/app_music.C:121`

Automatically continue playing if the reason for muting is gone..

If 'true', the music player resumes playing if the 'mute' resource changes from 1 to 0. If 'false', the player stays paused. The latter may be useful if there are multiple phones in the room, and the user answers with a phone other than that controlling the player.

8.7.8. Parameters of Domain `var.music`

`var.music.server` (string) `wallclock/app_music.C:130`

MPD server to connect to first.

8.7.9. Parameters of Other Domains

`sync2l` (bool) [= false] `wallclock/system.C:59`

Enable a Sync2l interface to the device's address book via a named pipe.

If enabled, a named pipe special file 'HOME2L_ROOT/tmp/sync2l' is created via which the device's address book can be accessed by the "sync2l" PIM synchronisation tool. If you do not know that tool (or do not use it), you should not set this parameter.

The pipe is created automatically, and it is made user and group readable and writable (mode 0660, ignoring an eventual umask). It is recommended to set the SGID bit of the parent directory and let it be owned be group 'home2l', so that a Debian or other chroot'ed Linux installation can access the pipe.

`android.autostart` (string) `wallclock/system.C:228`

Shell script to be executed on startup of the app.

If defined, the named shell script is started and executed in the background on each start of the app. The path name may either be absolute or relative to HOME2L_ROOT. If the name starts with '!', the script is started with root privileges using 'su'.

It is allowed to append command line arguments.

`home2l.unconfigured` (bool) [= false] `wallclock/home2l-wallclock.C:34`

Print information for new users.

If set, the WallClock app shows an info box on startup indicating that this installation is still unconfigured and how it should be configured. This option should only be set by the factory config file.

8.8. List of Exported Resources

`ui/standby` (bool,wr) [= false] `wallclock/system.C:118`

Report and select standby mode.

If 'true', the screen is on, but eventually with reduced brightness (unless `ui/active` is also set). If neither `ui/active` nor `ui/standby` is 'true', the screen is switched off, and the device may enter a power saving mode, depending on the OS platform.

`ui/active` (bool,wr) [= false] `wallclock/system.C:127`

Report and select active mode.

If 'true', the screen is on at full brightness (as during interaction).

`ui/dispLight` (percent,ro) `wallclock/system.C:134`

Display brightness.

`ui/luxSensor` (float,ro) `wallclock/system.C:137`

Light sensor output in Lux.

`ui/mute` (bool,wr) [= false] `wallclock/system.C:141`

Audio muting.

If 'true', the music player is paused. This can be used to mute playing music if the doorbell rings or some other event in the house occurs which requires the attention of the user.

`ui/bluetooth` (bool,wr) `wallclock/system.C:150`

Report and set Bluetooth state.

`ui/bluetoothAudio` (bool,ro) `wallclock/system.C:153`

Report whether an audio device is connected via Bluetooth.

9. *DoorMan* – A Doorbell and Doorphone Service

9.1. Overview

DoorMan is a doorbell and doorphone service operated on the command line or as a background service. It must be linked with an IP phone library (presently *Liblinphone*, *PJSIP* support is planned).

9.2. List of Configuration Parameters

`doorman.<ID>.enable` (bool) [= false] `doorman/home2l-doorman.C:29`

Define and enable a door phone with ID <ID>.

`doorman.<ID>.linphonerc` (string) `doorman/home2l-doorman.C:33`

Linphone RC file for door phone <ID> (Linphone backend only).

With the Linphone backend, all settings not directly accessible by Home2L settings are made in the (custom) Linphone RC file.

`doorman.<ID>.register` (string) `doorman/home2l-doorman.C:39`

Phone registration string.

`doorman.<ID>.secret` (string) `doorman/home2l-doorman.C:42`

Phone registration password.

`doorman.<ID>.rotation` (int) [= 0] `doorman/home2l-doorman.C:45`

Phone camera rotation.

doorman.<ID>.buttonRc (string) doorman/home21-doorman.C:49

External resource representing the bell button (optional; type must be 'bool').

There are two options to connect to a door button, which is either by defining an external resource using this parameter or by using the internal resource doorman-ID/button. If the external resource is defined, both resources are logically OR'ed internally.

doorman.<ID>.buttonInertia (int) [= 2000] doorman/home21-doorman.C:57

Minimum allowed time (ms) between two button pushes (default = 2000).

Button pushes are ignored if the previous push is less than this time ago.

doorman.<ID>.dial (string) doorman/home21-doorman.C:63

Default number to dial if the bell button is pushed.

doorman.<ID>.openerRc (string) doorman/home21-doorman.C:67

External resource to activate if the opener signal is received (optional).

doorman.<ID>.openerDuration (int) [= 1000] doorman/home21-doorman.C:70

Duration (ms) to activate the opener.

doorman.<ID>.openerHangup (int) [= 0] doorman/home21-doorman.C:73

Time (ms) after which we hangup after the opener was activated (0 = no hangup).

9.3. List of Exported Resources

doorman-ID/button (bool,wr) [= false] doorman/home21-doorman.C:158

Virtual bell button of the specified doorphone.

Driving this resource to true or false is equivalent to pushing or releasing a door bell button. To trigger a bell ring, a push and release event must occur. The "ID" in the driver name is replaced by the name of the declared phone.

There are two options to connect to a door button, which is either by defining an external resource using this parameter or by using the internal resource `doorman.<ID>.buttonRc`. Internally, both resources are logically OR'ed.

`doorman-ID/dial` (string,wr)

`doorman/home2l-doorman.C:172`

Number to dial for the specified doorphone.

This is the number dialed if the door button is pushed. The default value is set to the configuration parameter `doorman.<ID>.dial`. This resource allows to change the number to dial dynamically, for example, in order to temporarily redirect door bell calls to a mobile phone when out of home. The "ID" in the driver name is replaced by the name of the declared phone.

10. Driver Library

This chapter is the reference documentation of all supplied drivers. The driver library is still under construction. Drivers planned to be implemented next are:

- **MQTT:** Talk to MQTT brokers and devices by the MQTT protocol.
- **Service:** System services (SysV / systemd) suitable to export any system service as a resource, for example, Wifi access points (hostapd) or video recorder services (vdr).

10.1. Driver *Signal* (built-in)

The *Signal* driver is always available and allows to declare resources which just report back any driven value without any technical functionality.

Signals can serve as intermediate resources or for testing purposes.

They can be defined inside a `resources.conf` configuration file (see Section 5.4) or by the API calls `RcRegisterSignal()` (C/C++) or `NewSignal()` (Python).

10.2. Driver *Timer* (built-in)

10.2.1. Description

The *Timer* driver provides resources reflecting the current time, triggers to initiate hourly or daily tasks, and a set of resources reflecting day and night times, suitable, for example, to control an automatic outdoor light.

The driver is statically built into the *Resources* library and enabled by default. It can be disabled by the `rc.timer` setting.

10.2.2. Exported Resources

`timer/twilight/day` (bool,ro) `resources/rc_drivers.C:91`

Flag to indicate day time (time between official sunset and sunrise).

timer/twilight/day06 (bool,ro) resources/rc_drivers.C:94

Flag to indicate civil day time (time between civil dawn and dusk).

timer/twilight/day12 (bool,ro) resources/rc_drivers.C:97

Flag to indicate nautical day time (time between nautical dawn and dusk).

timer/twilight/day18 (bool,ro) resources/rc_drivers.C:100

Flag to indicate astronomical day time (time between astronomical dawn and dusk).

timer/twilight/sunrise (time,ro) resources/rc_drivers.C:104

Today's official sunrise time.

timer/twilight/dawn06 (time,ro) resources/rc_drivers.C:107

Today's civil dawn time.

timer/twilight/dawn12 (time,ro) resources/rc_drivers.C:110

Today's nautical dawn time.

timer/twilight/dawn18 (time,ro) resources/rc_drivers.C:113

Today's astronomical dawn time.

timer/twilight/sunset (time,ro) resources/rc_drivers.C:117

Today's official sunset time.

timer/twilight/dusk06 (time,ro) resources/rc_drivers.C:120

Today's civil dusk time.

timer/twilight/dusk12 (time,ro) resources/rc_drivers.C:123

Today's nautical dusk time.

timer/twilight/dusk18 (time,ro) resources/rc_drivers.C:126

Today's astronomical dusk time.

timer/now (time,ro) resources/rc_drivers.C:268

Current time (updated once per second).

timer/daily (trigger,ro) resources/rc_drivers.C:272

Triggers once per day (shortly after midnight).

timer/hourly (trigger,ro) resources/rc_drivers.C:275

Triggers once per hour (at full hour).

timer/minutely (trigger,ro) resources/rc_drivers.C:278

Triggers once per minute (at full minute).

10.3. Driver *GPIO*

10.3.1. Description

The *GPIO* driver is a universal driver leveraging the Linux *sysfs* GPIO capabilities to access general purpose inputs and outputs (GPIO).

In order to allow GPIOs to be used from a normal user application, they must be set up properly beforehand. This preparation requires *root* privileges and is therefore done by the *Home2L* init script at boot time. The names and configurations (e. g. direction, initial value) of available GPIOs are defined by symbolic links residing in

```
$HOME2L_ROOT/etc/gpio.<machine name>
```

pointing to the actual device, typically:

```
/sys/class/gpio/gpio<n>
```

The links are read both by the *GPIO* driver and the init script, and they must conform to the following naming conventions:

```
$HOME2L_ROOT/etc/gpio.<machine name>/<port name>.<options>

<options> is a sequence of characters and may include:
    i - The port is an input.
    0 - The port is an output with a default value of 0.
    1 - The port is an output with a default value of 1.
    n - The port is active-low (negated).
```

10.3.2. Exported Resources

The exported resources depend on the configuration (see above). They are named after their port names (*gpio/<port name>*) as specified in the name of the symbolic link.

10.4. Driver *Brownies*

10.4.1. Description

The *Brownies* driver is a universal driver continuously polling a *Brownie* bus tree and exporting resources for all features of all *Brownies* declared in the database.

For its operation, a database (`brownies.conf`) is required, which defines the set of *Brownies* to be polled. Devices without a valid and consistent database entry are ignored by the driver. Details on creating, updating and verifying the database can be found in Section 7.6.

The driver supports a single *Brownie* tree associated with a single *i2c* root master interface of the host machine. Multiple trees connected to the same Linux machine via multiple *i2c* interfaces can be driven by running separate *Home2L* instances for each interface, each running an instance of this driver with an individual database file.

Note: This driver cannot be hosted by `home2l-brownie2l`, which contains its own built-in version of a *Brownie* driver.

10.4.2. Configuration Parameters

`br.database` (string) [= "brownies.conf"] `brownies/brownies.C:48`

Name of the Brownie database file (relative to the etc domain).

br.link (string) [= "/dev/i2c-1"] brownies/brownies.C:52

Link device (typically i2c) for communicating with brownies.

The path is absolute or relative to the Home2L 'tmp' directory. In practice, the path will either point to a real i2c device (path is absolute) or to a maintenance socket of another Home2L instance on the same machine (path may be relative). If the special string "=" is given, the socket specified by br.serveSocket is used.

Supported i2c devices are Linux i2c devices and the 'ELV USB-i2c' adapter. The type is auto-detected.

br.serveSocket (string) brownies/brownies.C:65

Maintenance socket for the Brownie driver.

If set, the Brownie2L ('home2l-brownie2l') can connect to a running driver and use its link for maintenance and viewing statistics. During the time of the connection, the driver will pause all own link activities.

The path is absolute or relative to the Home2L 'tmp' directory.

br.checksPerScan (int) [= 1] brownies/brownies.C:75

Number of devices polled completely per fast scan.

Increasing this value will increase the general polling frequency of Brownie devices at the expense of a decreased the responsiveness on events with notifications (e.g. button events or switch sensor events).

As a rule of thumb, this value should be set such that the average times for the "fast polling phase" and the "slow polling phase" shown in the link statistics are in the same order of magnitude.

br.minScanInterval (int) [= 64] brownies/brownies.C:87

Minimum polling interval [ms].

Minimum time between starting two scans of the Brownie bus by the driver. If scanning all devices takes less than this, the next scan will be delayed. This avoids a high CPU load if only few or no devices are present.

br.featureTimeout (int) [= 5000] brownies/brownies.C:95

Time after which an unreachable feature resource is marked invalid.

br.temperatureInterval (int) [= 5000] brownies/brownies.C:99

Approximate polling interval for temperature values.

br.shades.requestAttrs (string) brownies/brownies.C:103

Request attributes for requests generated on button pushes [rc.userReqAttrs].

If a shades button is pushed, a request is auto-generated (or removed) to let the shades move up or down. This parameter defines the attributes of such requests. The most useful attribute is the off-time. For example, if the attribute string is "-31:00" and a user pushes a button to close the shades, this overrides automatic rules until 7 a.m. on the next morning. Afterwards, automatic rules may open them again.

By default, the value of rc.userReqAttrs is used.

Note: An eventual off-time attribute is set only after the button is released.

br.shades.defaultPos (int) [= 0] brownies/brownies.C:119

Default position for shades.

If set and >= 0, a default request for 'shades/pos' resources is set automatically. To disable the default request, enter a negative value here.

br.matrix.win.<brownieID>.<winID> (string) brownies/brownies.C:126

Define a window state resource.

This defines a resource representing a window state (type 'rctWindowState') based on a set of one or two sensor elements. The syntax of a defintion is

s:<sensor>
: Single sensor (0 = window open, 1 = window closed).

v:<lower>:<upper>
: Two sensors mounted at the side border of the window. Both sensors = 0 indicate that the window is open, only the upper = 0 is interpreted as a tilted window.

h:<near>:<far>:<tth>
: Two sensor mounted at the top border of the window, placed at the near and far end related to the hinge. Whether the window is open or tilted is determined dynamically by the order the switches open. 'tth' is the time threshold in ms. If the near sensor opens less than this later than the far sensor, the window is considered to be tilted. Otherwise, it is considered open.

The sensors are identified by 2-digit numbers as the raw matrix ids.

Note for the horizontal ('h') variant: This is the dynamic case, where both the near and the far sensor typically open both the "tilted" and "open" case. When tilting, the sensors typically open approximatly at the same time (either may be slightly earlier, this may change). However, when opening, the near sensor may open approx. 500-1000 ms later. Certainly, this time may vary depending on the window handling and properties of the sensors. Hence, horizontal placement should be avoided as possible when placing the sensors.

`brownie2l.historyFile` (string) [= ".brownie2l_history"] `brownies/home2l-brownie2l.C:46`

Name of the history file for home2l-brownie2l, relative to the user's home directory.

`brownie2l.historyLines` (int) [= 64] `brownies/home2l-brownie2l.C:49`

Maximum number of lines to be stored in the history file.

If set to 0, no history file is written or read.

10.4.3. Exported Resources

`brownies/<brownieID>/gpio/<nn>` (bool,ro) `brownies/brownies.C:374`

Brownie GPIO (input).

<nn> is the GPIO number, possible numbers are those with the respective bit set in 'SBrFeatureRecord::gpiPresence'.

`brownies/<brownieID>/gpio/<kk>` (bool,wr) [= <preset>] `brownies/brownies.C:381`

Brownie GPIO (output).

<kk> is the GPIO number, possible numbers are those with the respective bit set in 'SBrFeatureRecord::gpoPresence'.

<preset> is the preset value as defined by 'SBrFeatureRecord::gpoPreset' an is set as a default.

`brownies/<brownieID>/temp` (temp,ro) `brownies/brownies.C:493`

Brownie temperature sensor value.

brownies/<brownieID>/matrix/win.<winID> (windowstate,ro) brownies/brownies.C:623

Brownie window state.

Reports a window state (closed/open/tilted) based on one or two matrix sensor switches. The window must be declared by a br.matrix.win.<brownieID>.<winID> configuration entry.

brownies/<brownieID>/matrix/<nn> (windowstate,ro) brownies/brownies.C:700

Brownie sensor matrix value.

<nn> is a two-digit number, where the first digit represents the row and the second digit represents the column of the respective sensor.

brownies/<brownieID>/shades<n>/pos (windowstate,ro) brownies/brownies.C:859

Brownie shades/actuator position.

Current position of an actuator. An 'rcBusy' status indicates that the actuator is currently active / moving.

<n> is the index of the actuator: 0 or 1 if the Brownie drives two actuators or always 0, if there is only one.

The driver issues automatic user requests if one of the buttons are pushed. The attributes of such requests are specified by br.shades.requestAttrs, rc.userReqId, and rc.userReqAttrs.

Whether a default request is set and its potential value are defined by br.shades.defaultPos.

brownies/<brownieID>/shades<n>/actUp (windowstate,ro) brownies/brownies.C:875

Brownie actuator is powered in the "up" direction.

This resource reflects the actual (raw) state of the actuator, and is 'true' iff the engine is presently powered in the "up" direction. This resource is read-only, to manipulate the actuator, a request must be issued for the brownies/<brownieID>/shades<n>/pos resource.

<n> is the index of the actuator (0 or 1).

brownies/<brownieID>/shades<n>/actDown (windowstate,ro) brownies/brownies.C:885

Brownie actuator is powered in the "down" direction.

This resource reflects the actual (raw) state of the actuator, and is 'true' iff the engine is presently powered in the "down" direction. This resource is read-only, to manipulate the actuator, a request must be issued for the brownies/<brownieID>/shades<n>/pos resource.

<n> is the index of the actuator (0 or 1).

brownies/<brownieID>/shades<n>/btnUp (windowstate,ro) brownies/brownies.C:895

Brownie actuator's "up" button is pushed.

This is the actual (raw) state of the actuator's "up" button.

<n> is the index of the actuator (0 or 1).

brownies/<brownieID>/shades<n>/btnDn (windowstate,ro) brownies/brownies.C:902

Brownie actuator's "down" button is pushed.

This is the actual (raw) state of the actuator's "down" button.

<n> is the index of the actuator (0 or 1).

10.5. Driver *Weather*

10.5.1. Description

The *Weather* driver provides local weather information by querying the Open Data service of the German Weather Service (DWD).

10.5.2. Configuration Parameters

weather.stationID (int) [= NULL)] drivers/weather/home21-drv-weather:57

Station ID according to the DWD station table.

See "MOSMIX-Stationskatalog", available for download at https://www.dwd.de/opendata .

weather.debug (int) [= False)] drivers/weather/home2l-drv-weather:67

Run the driver in debug mode.

In debug mode, radar eye images are not exported as a resource, but written into a local image file. This facilitates to run the driver directly on the command line for debugging purposes.

10.5.3. Exported Resources

weather/temp (temp,ro) drivers/weather/home2l-drv-weather:106

Outside temperature.

weather/pressure (float,ro) drivers/weather/home2l-drv-weather:109

Outside air presssure reduced to mean sea level.

weather/humidity (percent,ro) drivers/weather/home2l-drv-weather:112

Relative humidity.

weather/windDir (int,ro) drivers/weather/home2l-drv-weather:115

Mean wind direction during last 10 min. at 10 meters above ground.

weather/windSpeed (float,ro) drivers/weather/home2l-drv-weather:118

Mean wind speed during last 10 min at 10 meters above ground.

weather/windMax (float,ro) drivers/weather/home2l-drv-weather:121

Maximum wind speed last hour.

weather/weather (int,ro) drivers/weather/home2l-drv-weather:124

Present weather condition according to a table provided by the DWD..

See file 'poi_present_weather_zuordnung.pdf', available for download at https://www.dwd.de/opendata .

`weather/radarEye` (string,ro) `drivers/weather/home2l-drv-weather:225`

Radar Eye.

An .pgm-encoded image of 128x128 pixels showing the weather radar around the own position configured by as configured by the 'location.*' settings together with the wind direction and strength.

The cross indicates the ego position and is shifted away from the center such that the wheather indicated in the center of the radar eye will reach the own position in 2 hours. The cross has a radius of 50 km.

`weather/radarWarning` (bool,ro) `drivers/weather/home2l-drv-weather:237`

Radar Warning (NOT IMPLEMENTED YET).

Flag indicating that it may be raining and any roof windows should be closed.

A. Appendix: Documentation How-To

A.1. Writing the *Home2L Book*

A.1.1. Formatting

`\code{foo_bar}`
: Print as code, not specially highlighted, but '_' characters are allowed without escaping. *Note: Spaces are not allowed in the text!*

`\lst{foo_bar}`
: Inline listing, with the rules of `\code{}` applied. *Note: Spaces are not allowed in the text!*

`\lstf{foobar}`
: Inline listing formattable. LaTeX formatting is allowed, but '_' characters must be escaped, for example.

`\begin{lstlisting}` **...** `\end{lstlisting}`
: Display listing: full text witdh, with auto-wrapping. The following languages are supported (`\begin{lstlisting}[language=<language>]`):

  (default): Plain, all text black.

  comments: All text black, only comments starting with `#` are grey.

  bash: Bash session: Lines starting with `\$` (commands) black, output and comments grey. Command lines can be prefixed with "§" explicitly.

  python: Python session: Lines starting with `>>>` (commands) black, output and comments grey. Command lines can be prefixed with "§" explicitly.

  home2l: `home2l-shell` session: Commands black, output grey.

  brownie2l: `home2l-brownie2l` session: Commands black, output grey.

`\lstbox{foobar}`
: Display listing, for places where the former are not allowed, e. g. in info boxes

A.1.2. Info and Warn Boxes

`infobox{multi-line text}`
: Print an info box. Info boxes contain supplemental information, not required for all readers.

`warnbox{multi-line text}`
: Print a warn box. Such boxes contain important information, requiring special attention from the reader.

A.1.3. Internal References: Tools, Configuration, Resources

A.1.3.1. Labeling

`\labeltool{home2l-newtool}`
: Set section label to define a new tool.

`\idx{term}`
: Add and link word "term" to the index, while printing it as-is in the text.

A.1.3.2. Referencing

`\reftool{home2l-sometool}`
: Reference a tool.

`\refenv{domain.some.variable}`
: Reference the environment parameter `domain.some.variable`.

`\refrc{driver/local/id}`
: Reference a resource by its host-local ID.

A.1.4. External References: Source files, Doxygen Pages, Internet

`\refdoc{path/to/file}{written_text}`
: Add a hyperlink to some document in the source tree. The file name is given relative to the source tree. The "written text" appears without special formatting. Typical use: Referencing some other PDF document such as `README.pdf` or some Doxygen HTML document.

`\refsrc{path/to/file}`
: Add a hyperlink to some source file. The filename is written in typewriter font.

`\href{url}{written_text}`
: Add a hyperlink to some internet URL.

`\refapic{name}`
: Add a reference to some C/C++ class, function, or other type of declaration. Presently, this prints the name of the object ("`name`") and links it to the main page of the C/C++ API. The name can be entered into the "search" field of the Doxygen internal search.

`\refapicgroup{group__subgroup}{codedocumentation(Group/Subgroup)}`
: Add a reference to a Doxygen group in the C/C++ API as declared by '@newgroup'. The second argument is an example for the written text.
Note: Underscores must be doubled in the first argument!

`\theapic{}`
: Print the text "C/C++ API" with a hyperlink to the main page of the C/C++ API documentation.

`\refapipython{name}`
: Add a reference to some Python function or other type of declaration. Presently, this prints the name of the object ("name") and links it to the main page of the Python API. The name can be entered into the "search" field of the Doxygen internal search.

`\refapipythongroup{group__subgroup}{codedocumentation(Group/Subgroup)}`
: Add a reference to a Doxygen group in the Python API as declared by '@newgroup'. The second argument is an example for the written text.
Note: Underscores must be doubled in the first argument!

`\theapipython{}`
: Print the text "Python API" with a hyperlink to the main page of the Python API documentation.

A.2. Documenting Configuration Parameters

Configuration parameters are documented by means of a specially formatted comment following the respective `ENV_PARA_...` macro defined in `common/env.H`. The comment has the following structure:

```
ENV_PARA_... (...)
  /* Short description (max. ~40 chars, no trailing period)
   *
   * Optional long description, optionally covering multiple lines or empty lines.
   * Only the last line must not be empty, and there must be exactly one empty
   * line between the short and the long description.
   *
   * Formatting can be done with LaTeX syntax.
   */
```

In the description part, any formatting or cross-referencing macros described in Section A.1 can be used.

A.3. Documenting Resources

Resources are documented by special comments following the respective `RcRegisterResource()`, `CResource::Register()`, or `CRcDriver::RegisterResource()` invocations:

```
rcMyFirst = RcRegisterResource (drv, "myFirstLid", rctBool, false);
  /* [RC:mydriver] Some read-only Boolean resource
   */
```

```
rcMySecond = drv->RegisterResource ("mySecondLid", rctPercent, true);
rcMySecond->SetDefault (0.0);
  /* [RC:mydriver] Some writable resource with a percentage value type and a default ↩
    ↪request
   *
   * Optional long description, optionally covering multiple lines or empty lines.
   * Only the last line must not be empty, and there must be exactly one empty
   * line between the short and the long description.
   *
   * Formatting can be done with LaTeX syntax.
   */
```

The following conditions must be met:

- If a default value is to be set, the respective `SetDefault()` call must follow the registering call in the next line.
- The next line must be a comment of the form `/* [RC:<driver>] <brief description>`.
- The `[RC:<driver>]` clause defines the driver.
- Other information – the Resource's local ID (LID), its type, an the writability flag are extracted from the registering call. If the LID and/or default value are not written explicitely in these lines (e.g. they are calculated in variables), they can be appended to the `[RC]` clause: `[RC:<driver>:<lid>]...` or `[RC:<driver>:<lid>:<default>]...`. Examples can be found in `brownies/brownies.C`.
- The comment block must end with a line starting with "*/" (end of comment).
- To explicitly exclude the resource from documentation, place a line with `/* [RC:-] */` behind the registering call (either in the same or the following line).

The following characters are automatically escaped and may appear without escaping in the text: '_', '#', '$', '%'.

As for the documentation of configuration parameters, the short description has no trailing period, and in the description part, any formatting or cross-referencing macros described in Section A.1 can be used.

A.4. Doxygen Cheat Sheet

A.4.1. Formatting

In Doxygen comments, Markdown formatting can be used.

1. Enumerations:

```
/// The type 'T' must fullfill the following properties:
///
/// 1. It must implement a method 'const char *ToStr ()' for the 'Dump' method.
///    (the returned string will not be used after a further call to this method
///    for any object, so that it can e.g. be stored in a static variable.)
///
/// 2. A second item may follow here. Empty lines before items (also before the
///    first) have no effect and can be left out.
///
```

2. Itemize:

```
/// The type 'T' must fullfill the following properties:
///
/// - It must implement a method 'const char *ToStr ()' for the 'Dump' method.
///   (the returned string will not be used after a further call to this method
///   for any object, so that it can e.g. be stored in a static variable.)
///
/// - A second item may follow here. Empty lines before items (also before the
///   first) have no effect and can be left out.
///
```

3. Code:

```
/// @code
/// #if WITH_FOOBAR == 1
/// APP(Foobar, "foobar")
/// #endif
/// @endcode
```

A.4.2. Referencing Code

`@ref CSomeClass`
: Reference a C++ class

`[*Home2L Book*](../home2l-book.pdf)`
: Reference the *Home2L Book*

A.4.3. Other Hints

`@internal`
: Exclude item (e.g. C macro, but no class member) from the output.

`@private`
: Exclude a class method or global variable from the output.

Index